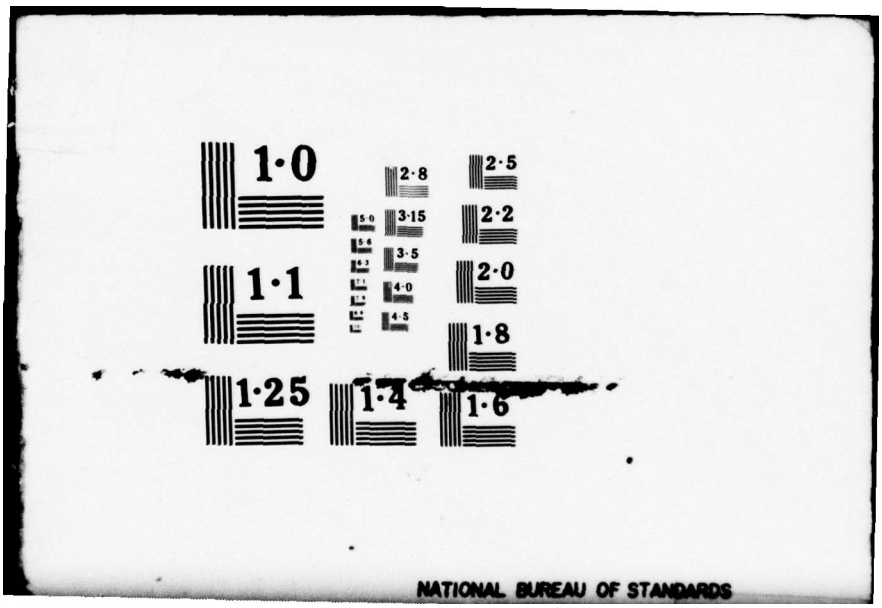
AD-A055 945 ARMY MILITARY PERSONNEL CENTER ALEXANDRIA VA F/G 9/2
X-SIM. A DYNAMIC COMMUNICATIONS SIMULATOR FOR THE X-TREE NETWOR--ETC(U)
JUN 78 P A SUHLER

UNCLASSIFIED NL

| OF |
ADA
065945

END
DATE FILMED
8-78
DDC


1·0
1·1
1·25
2·8
3·15
3·5
4·0
4·5
1·4
2·5
2·2
2·0
1·8
1·6
NATIONAL BUREAU OF STANDARDS

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

| REPORT DOCUMENTATION PAGE | READ INSTRUCTIONS BEFORE COMPLETING FORM |
|---|---|
| 1. REPORT NUMBER / 2. GOVT ACCESSION NO. | 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE (and Subtitle)<br>XSIM: A Dynamic Communications Simulator for the X-Tree Network Report and User's Manual | 5. TYPE OF REPORT & PERIOD COVERED<br>Final Report<br>15 June 1978<br>6. PERFORMING ORG. REPORT NUMBER |
| 7. AUTHOR(s)<br>Suhler, Paul Arthur 462-04-2993<br>1LT, EN | 8. CONTRACT OR GRANT NUMBER(s) |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS<br>Student, HQDA, MILPERCEN, (DAPC-OPP-E)<br>200 Stovall Street<br>Alexandria, VA 22332 | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS |
| 11. CONTROLLING OFFICE NAME AND ADDRESS<br>HQDA, MILPERCEN, ATTN: DAPC-OPP-E<br>200 Stovall Street<br>Alexandria, VA 22332 | 12. REPORT DATE<br>15 June 1978<br>13. NUMBER OF PAGES<br>86 |
| 14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) | 15. SECURITY CLASS. (of this report)<br>Unclassified<br>15a. DECLASSIFICATION/DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

This is a Master's degree research report from the Department of Electrical Engineering and Computer Sciences, the University of California, Berkeley. It is part of the on-going X-Tree research project.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

computer networks, multiprocessors, virtual channels, performance measurement

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report introduces the virtual channel concept of interprocessor communications and describes the information needed to evaluate its performance in X-Tree. The requirements for a dynamic communications simulation are derived and an implementation is presented which simulates to the level of byte transmission and produces a trace output. An example

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

statistics program is given which inputs the trace and outputs a statistical summary of selected performance indices for the network. A user's guide for the simulator is included as are sample simulation outputs.

# DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY PRACTICABLE. THE COPY FURNISHED TO DDC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

# XSIM: A Dynamic Communications Simulator for the X-Tree Network

1LT Paul A. Suhler
HQDA, MILPERCEN (DAPC-OPP-E)
200 Stovall Street
Alexandria, VA 22332

Final Report

15 June 1978



A Research Report Submitted to
the Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
in partial fulfillment of the requirements for the degree of
Master of Sciences, Plan II.

(6)

# X-SIM,

# A DYNAMIC COMMUNICATIONS SIMULATOR FOR THE X-TREE NETWORK

## REPORT AND USER'S MANUAL.

(9) Final rept.,

by

(10) Paul Arthur/Suhler

(11) 15 Jun 78 (12) 99 P.

RESEARCH PROJECT

Submitted to
the Department of Electrical Engineering and Computer Sciences,
University of California, Berkeley,
to partial satisfaction of the requirements for the degree of
Master of Sciences, Plan II.

Approval for the Report and Comprehensive Examination:

COMMITTEE: _____ Research Advisor

June 15, 1978 Date

D. A. Patterson

June 15, 1978 Date

ABSTRACT

This report introduces the virtual channel concept of interprocessor communications and describes the information needed to evaluate its performance in X-Tree. The requirements for a dynamic communications simulation are derived and an implementation is presented which simulates to the level of byte transmission and produces a trace output. An example statistics program is given which inputs the trace and outputs a statistical summary of selected performance indices for the network. A user's guide for the simulator is included as are sample simulation outputs.

ACKNOWLEDGEMENTS

I am grateful to a number of people who have helped me during my stay at Berkeley. Professor Alvin Despain has provided constant encouragement and guidance in the course of this project. Professor David Patterson offered advice on the requirements and implementation of the simulation. Finally, I wish to thank the U. S. Army for sponsoring my graduate study under the Army Fellowship Program.

TABLE OF CONTENTS

# 1 INTRODUCTION

The X-Tree research project [1] is an attempt to design a computer system composed of identical single-chip computers (X-Nodes) connected in a hierarchical structure. Each computer has a processor with a local memory and is connected to the network via an intelligent communications switch having ports attached to a small number of adjacent nodes.

The network has the basic form of a binary tree, although at least one additional link to the network is provided to shorten inter-computer message paths and to allow fault-tolerance. As the network is regular, each node can be assigned a unique number and a message transmitted through the network with all routing done locally in each communications switch on the path.

The specific network structure will be chosen based upon several criteria:

(1) Minimum message path length.

(2) Minimum number of ports per node.

(3) Existence of a locally-executable routing algorithm.

A large number of structures have been studied to determine the average path length when every node transmits a message to every other node [2]. The next step, following this "static" simulation, is to simulate the operation of the network in detail, acounting for message interference. This requires including the effects of the communications switch architecture, the communications protocols, and failures of nodes and links.

This report concerns the development and implementation of

a simulation program for such a study of the dynamic operation of the network. The emphasis is on the design of the program, with simulation results used as examples of its operation. This report is organized into sections concerning: objectives and requirements of the study (2), simulator design considerations (3), the details of the implementation (4), validation procedures (5), and a user's manual (Appendix A).

Results of studies using this simulator will be used in further designs of the network, the communications switches, and routing algorithms. The simulator may be extended to implement execution of toy programs for operating system studies.

Except for constructing and measuring an actual system, simulation is the best way to determine the behavior of the system. Thus, as much simulation as possible will be done before building a prototype X-Tree. There will, however, eventually come a point at which simulation to the required degree of detail becomes too expensive and an actual system must be built in order to learn more.

# 2 REQUIREMENTS

## 2.1 System Description

The X-Tree network consists of a number of computers connected via bi-directional communication links in a binary tree structure. As Figure 1 shows, additional links may be added; the structure shown is called a "full-ring" tree.

For purposes of simulation, the essential part of the X-Tree network is the communications switch which is a component of every X-Node (Figure 2). This switch enables messages to be transmitted through the network without the attention of the CPUs along the paths. The requirements for the simulator can be derived by examining the operation of the network in general and of the switch in particular.

2.1.1 Messages. Messages in X-Tree are communications between processes residing in different processors. The messages may be long, as with virtual memory pages sent from a disk at the bottom of the tree to a node, or short, as with requests for those pages. One format has been devised to cover all messages; this format has routing and control information separate from the data. The routing and control function is built into the switches, which can then pass the data without examining it.

Each message is prefaced by a header byte and a number of address bytes (Figure 3). When a communication switch receives a

new message over one of its links, it can inspect the address and perform a routing algorithm to determine which link to send it out. An arbitrary number of data bytes can then follow the address. Finally, a "teardown" byte will terminate the message.

2.1.2 Virtual channels. Many messages may share the same link(s) for part of their paths. If a message retains total control of a link until it completes, then blocking that message blocks all the links on its path; deadlocking of the network can then occur. To allow use of a link even though a message through it is stalled, the virtual channel has been devised. The basic approach is to buffer parts of several messages on each end of a link; when one message stalls, another is transmitted. The virtual channel is similar to the "circuit" defined by Tyme [3], except that it is unidirectional.

A virtual channel would be established for messages sent from one process to another. When no more messages are to be sent, the channel is released. In terms of Figure 3, the header and address establish the virtual channel, the data blocks would be separate messages, and the teardown byte removes the channel.

2.1.3 Communications switches. Each communications switch has several ports among which messages are passed; one port goes to the node's CPU, the other ports to adjacent nodes. There is also a communications controller (a simple processor), to route messages and control the ports. All of these are connected via a high-speed bus which permits full communication while preventing simultaneous access to the same port.

Each port has two separate buffer structures and finite state machines for control: one is for messages coming in from the link and going over the bus to another port; the other is for messages from the bus going out across the link (Figure 4). Each buffer has a fixed number of words for each of several physical channels. Bytes sent across a link from the buffer of one channel are stored in the buffer of the same channel on the other side. Communication in the opposite direction is the same for those buffers. The bytes sent across the bus are stored into a different physical channel's buffer at the output port. The output port and channel had been previously selected by the controller which ran the routing algorithm and then selected an unused channel at that port. Thus, a virtual channel through the network will go through different physical channels at each link.

2.1.4 Controller operation. When a message arrives at an input port through an unused channel, it is stopped until the controller can service it. The first part of the message, the header and address bytes, are then sent into the controller as they arrive. When all of the address has arrived, the controller can perform the routing algorithm. It then selects an unused physical channel at that port and transmits the header and address to it. When that is finished, the controller modifies a control table at the input port to cause that channel to be sent directly to the selected output port and channel.

2.1.5 Input port operation. An input port will thus have several active physical channels. The interport bus is run on a

fixed time slot basis: once each time around, each input port will be able to transfer a data byte from one channel's buffer to the output port. The task of the finite state machine at the port is then to select a channel for intra-node transmission. Generally, it will send consecutive bytes from one channel until either the buffer runs dry or the output buffer fills up and rejects a byte. The FSM then goes to the next channel with data in its buffer.

An alternative strategy would be to go to a new channel after each byte, to avoid overflowing the receiving buffer. Both of these techniques will be investigated.

2.1.6 Output port operation. The output ports function very similarly, selecting a channel for transmission and then sending as many bytes as possible. The difference is that since only one byte can be sent at a time across the link, the channel number cannot be sent in parallel with the data, as it can across the bus. Thus, when beginning a new channel, the channel number is sent first and then the data bytes.

2.1.7 Link protocol. The two output ports sharing a link may both desire to send at the same time. A small amount of logic and some control lines can resolve this contention; the effect is that when both have data to send, they will alternate byte by byte. This is independent of whether a channel number or data byte is being sent.

2.1.8 Channel teardown. When one process no longer desires to

send to another it can transmit a special teardown byte along the virtual channel. Each input port recognizes the byte and, after transmitting it to the output port, disables the physical channel so that the next data received on it is sent to the controller.

In summary, processes communicate via unidirectional virtual channels which they set up when needed and tear down when no longer needed. These channels are set up by sending a header and address from source to destination. At each node on the path, a routing algorithm is executed and a pointer established from the physical channel at the input port to the physical channel at the output port. As many data bytes as desired are then sent along the virtual channel. Finally, a teardown byte is sent, removing the links between physical channels, thus removing the virtual channel.

## 2.2 Objectives

The purpose of the dynamic simulation is to determine the feasibility of the X-Tree concept in general and of the communications scheme in particular The success of X-Tree will largely depend upon its ability to transmit internode messages rapidly. The design of the communications switch will thus require accurate statistical information about the performance of various structures under different workloads.

2.2.1 Design information. The design of simgle-chip computers is in some sense reversing the trend of recent years toward cheap

hardware. On the chip surface, the silicon (or whatever material) "real estate" is precious; the maximum function must be obtained from a limited area. Off-chip communication is now relatively expensive: drivers consume much area and are slow. The pins also represent a prime source of failures.

This leads to some design principles: off-chip communication must be minimized (aided by powerful processing on-chip), pinouts must be kept as few as possible, and the on-chip area dedicated to communications must be minimized. These in turn suggest the classes of information which the dynamic simulation must deal with.

The message buffers within the switch should be minimized to allow more area for the CPU. This problem has (literally) two dimensions: the number of bytes per channel gives the width of the buffer and the number of channels gives its height. If only a small number of channels in each port is used, the buffer height can be limited. If few of the enabled channels are ever busy, then the interprocess communication algorithms can be designed to release idle channels more frequently. Finally, if buffering only a small number of bytes per channel causes little degradation, the buffer length can be reduced.

Message routing algorithms need to be tested under conditions of faults and congestion. Algorithms for the same structure should be carefully compared before one is designed into the silicon.

2.2.2 Specific data. Designing the system will require comparing the performance of different structures under various

workloads. Simulations will yield performance indices for different system and workload variables. System variables are the network structure, routing algorithm, buffer sizes, and fault patterns. Structures will be essentially binary trees with one or two extra links per node. Various interconnection schemes can be devised for the extra links, and each scheme must have a routing algorithm. Buffer size includes both the number of physical channels and the buffer lengths. Lengths might not be the same for both input and output ports. Faults include both broken links and disabled nodes.

Workload variables reflect the lengths, generation frequency, and paths of the messages sent within the network. Page traffic, between leaves and other nodes, will be long, relatively infrequent messages. Short, more common, interprocess messages may be sent between any nodes. Message generation will eventually be determined by execution of a trace program, where receiving a message alters the state of a node, changing its generation pattern. Such a program would be designed to generate message traffic similar to that of an actual program.

Performance indices indicate how well messages flow and the degree of hardware utilization. The basic index is the total transmission time of a message with respect to its length, path, and the loading of the network. The number of consecutive bytes transmitted between rejections is a measure of the degree of message interference. The number of active channels and the buffer occupancy give the hardware utilization. Finally, the distribution of traffic over the network should be studied to identify bottlenecks.

## 2.3 Requirements

The key to the implementation of the simulator is the flexibility to permit accurate measurement for widely different conditions. The representation of system and workload variables must allow unforeseen additions while the representation of performance must allow measurement of new indices, all with minimum modification to the simulator.

The program must be easily expandable to permit adding new structures and routing algorithms which will certainly be devised after the simulator is written. More sophisticated message generators will be added, eventually to include toy program execution.

The simulator must be usable and modifiable without requiring that the user know its every detail. Finally, execution should be reasonably fast to permit long simulations.

# 3 DESIGN CONSIDERATIONS

This portion of the report enumerates and compares the possible methods of implementing the simulator. In the subsequent section on implementation, these considerations are applied to X-Tree in particular.

## 3.1 General

The most basic decision is the level at which to simulate the network. The least detailed is the implementation of some analytic function (or set of functions) describing a model of the network or some aspects of it. The most detailed would represent the complete state of each node to the level of programs in memory. The tradeoffs involve accuracy and validity versus run time and memory usage.

Implementation of an analytic model uses the least time and memory, but requires many simplifying assumptions. These assumptions limit the domain of validity of the model -- what is true for lightly-loaded networks may not hold when the traffic increases. In the case of the X-Tree network, models do not yet exist for any cases, although this simulator will aid in developing them.

Representation of the complete network state is not possible, either, given the uncertainties of the CPU design and the size of the network. Going to too great a degree of detail is not worthwhile, as the network will operate asynchronously, thus

invalidating an effectively synchronous simulation. However, if the detail is sufficient, then a variety of structures and workloads can be tested without affecting the validity. If the simulator functions like the actual network, then its operation can be traced as would that of an actual system.

## 3.2 Functional Partitioning

The functions performed by the simulator are: emulating the actual operation of the network, monitoring the operation, and statistically summarizing certain aspects of the operation. These can be carried out in essentially three different ways.

3.2.1 Data base approach. A common technique in performance measurement of existing computer system is to produce trace tapes to be studied off-line. In an analogous fashion, a simulator can produce a stream of trace messages to be stored in a data base [4]. If the trace is detailed enough, then the data from a single run will be useful for statistical studies of different performance indices, not all of which may be foreseen at the time the simulator is designed. The problem with the data base technique is that the amount of data produced can be enormous.

3.2.2 Direct analysis approach. The opposite approach is to intertwine the statistics processing with the actual simulation, and eliminate the trace output. This removes the need for a large data base and can produce the final data quicker and without having to coordinate the simulator and a statistics

program. However, this lacks flexibility: detailed reprogramming is needed to measure any but the standard indices. This revised program must then be retested.

3.2.3 Mixed approach. To avoid the size problem of the data base while retaining its flexibility, the trace can be fed directly into the statistics program, if the operating system under which the simulation is running permits this (e.g.: the UNIX pipe mechanism). As in the data base technique, the user must prepare the statistics program to accept a stream of standard trace messages and from them derive the desired information. The disadvantage is that the complete simulation must be rerun each time a new statistics routine is run.

## 3.3 User Interaction

During the initial debugging and validation and later when a new user is learning to use the simulator, it is convenient for it to run in an interactive manner. This requires prompting for input parameters and checking them for correctness as well as producing intelligible trace messages. Debugging tools should also be added to allow the user to see in detail the state of the simulation. Tools external to the simulator may also be used to measure code and data space and execution time.

There are a number of more specific issues which must be considered; these will be discussed in the next section, on implementation.

# 4 IMPLEMENTATION

This part of the report is concerned with the details of the implementation. References are made to portions of the simulator and statistics program code which are in Appendices B and C. With this information, a user should be able to modify the simulator to deal with any desired structure, workload, or statistical summary.

## 4.1 General

The simulator is a version 7 C program running on the U. C. Berkeley LECS Unix system on a PDP 11/70. It is a discrete-time simulation in which the time quantum, or "tick", represents the time for one byte to be sent across an internode link. This is also the time for one cycle through all the time slots of the interport bus within the communication switch. Rather than maintain a time-ordered event list, requiring the overhead to insert entries, the simulator at each tick searches through all the nodes, operating upon those that are active at that tick. This will be further explained below.

The simulation process is partitioned into a simulator which produces a trace of events in the network and a statistics-gathering program which reads the trace and compiles a summary of the network's behavior. While the simulator can be used to produce a data base, it is expected that most users will pipe the output directly into a statistics program to avoid

producing a large data base.

## 4.2 Simulator

The simulator has two phases: an initialization, in which the system and workload parameters are input, and the actual run, in which messages are moved and the trace is output. In each tick of the run, the simulator makes two passes across all of the nodes and ports. The first pass locates and marks ports having data to send and the second moves one byte from each of those ports. Separate passes are needed to prevent any byte from being moved more than one step in one tick.

4.2.1 Data representation. The data structures fall into four classes corresponding to the major elements of the sytem: nodes, ports, physical channels, and messages. An X-Tree is represented as an array of nodes, with each node having a controller, CPU, and an array of ports. Each port has an input and an output section, each having a pointer to a list of currently active channels. A channel has counters representing buffers on each end of a link; it would be conceptually more direct to put fixed-size arrays in each port to represent the multi-channel buffers. However, this would require much storage, of which relatively little would be active; the linked list approach conserves space. Finally, each message is passed by means of pointers; just as one actual message will occupy buffers in several channels, several simulated channels will have pointers to the same message. A typical link and its adjacent ports are shown in Figure 5a and

the corresponding simulated ports and channels in Figure 5b. The declarations of these structures are on page B-1 (of the simulator code).

4.2.1.1 Nodes. To represent the controller, each node has a counter "conbuf" of the number of address bytes in the controller's buffer. There is also a queue of channels waiting to send addresses to the controller; the queue is managed with pointers "waithead" and "waittail". The CPU is represented by the pointer "cpoutchnl" to the channel which is currently sending data over the link from the CPU. Conditions in the node include: node alive or dead, controller receiving, CPU sending channel number, and last byte from CPU rejected. These are encoded as one-bit flags in "nflag". This requires a total of 13 bytes.

Flags for both nodes and ports are encoded into character variables with one bit per item. A specific bit is tested by ANDing the flag with an octal constant. A bit is set by ORing with a constant and reset by ANDing with the complement of that constant. The formats of these flags are shown in Figure 6.

In the tree, nodes are numbered beginning with "1" (the root). As C numbers arrays beginning with "0", the first element of the node array will be unused. Thus, the constant "NODEMAX" is always one greater than the maximum allowable number of nodes.

4.2.1.2 Ports. The ports are numbered from zero to "portnum". Zero through two are the ports to parent, left child, and right child, respectively. Port "portnum" is the port to the

CPU. Other ports are defined according to the structure. The output section of each port has a pointer "outchnl" to the channel currently sending data across the link and a queue ("qhead" and "qtail") of messages awaiting a free channel. The input section has a pointer "inchnl" to the channel currently sending data across the bus. The counter "chnls" gives the number of output channels.

Status information in "pflag" tells: if the port has been checked by "passl", if the output is sending a channel number or a data byte, if the CPU is sending, and for both input and output, if the port is sending and if the last byte was rejected. The pointer "link" is to the port across the link, and "prnode" is to the node of which the port is a part; this node pointer speeds up references to other ports in the node. Each port requires 15 bytes.

4.2.1.3 Channels. Each channel has an output buffer "outbuf" and an input buffer "inbuf". These are checked for overflow against the global "outbuflim" and "inbuflim". There is a pointer "destc" to the output channel to which the input is sending; if the channel is waiting for the controller, this field is a link to the next channel in the waiting list. The pointers "nextc" and "lastc" are used to form the circular list of active channels. The input status is in "cflag"; the channel can be waiting for the controller, sending to the controller, or sending to an output port. This requires 11 bytes. Figure 7 illustrates the organization of channels in a node.

4.2.1.4 Messages. To be routed and passed properly, each message contains a destination node number, number of address bytes, total number of bytes, and teardown / regular indicator. To permit a useful trace, each also has a start time and a count of the number of communication switches it has passed through. A pointer to another message allows it to be queued. Each message structure is 12 bytes.

4.2.1.5 Descriptors. A final structure is the message descriptor, which contains the generation frequency, address and total lengths, and a path description. All nodes generating the same kind of message have a pointer to the same descriptor. Each descriptor is 13 bytes.

In each structure, character variables have been used when possible to reduce the size of the structure. This is of particular importance in the channels because of their large number.

4.2.2 Initialization. The system and workload variables are set by the user during execution of the "initialize" routine. The routine will prompt in English for parameters unless the "-s" (suppress prompt) argument is given when the simulator is called. Each parameter is read in and checked for validity; e.g., "nodenum", the number of nodes in the tree being simulated cannot be greater than "NODEMAX", the number of nodes in the array. The system variables specified are "nodenum", "inbuflim", "outbuflim", "shape" (the network structure), and "rtalgo" (the routing algorithm).

The structure is specified by setting the "link"s in each port. The links are set by using the function "otherport" which returns a pointer to the attached port. "otherport" uses the function "othernode" to find the number of the attached node. The number of ports is set according to the structure. Where it is possible to specify links algorithmically, the user should add a case to the switch to perform this.

For fault tolerance studies, nodes and links can be explicitly killed. A dead node has its alive flag set to zero and the links in adjacent nodes also set to zero. A dead link has "link" set to zero in both of the linked ports.

Workloads are specified by creating message descriptors and setting pointers to them in the nodes concerned.

Specific parameter input formats are given in the User's Manual, Appendix A.

4.2.3 Pass one. The procedure "pass1" is used to mark the ports which will be sending in the current tick. It is called from run" for each port which is not yet visited and which has a nonzero link. This test is put outside the procedure call to eliminate the calling overhead if no further action would occur.

In "pass1" both the current port and the port attached via the link are processed and then marked as visited. First, the input sections of the ports are processed as follows: if there is a channel on its list, then if there is more data to send and the last byte was not rejected, the port is set to send. Otherwise, the function "findinchnl" searches the list for the next channel with data; if one is found, the port is set to send; if

none with data is found, the port will not send. Finally, the reject flag is cleared.

The output port is processed in a similar manner, except that if a new channel is selected, the "sending channel number" flag is also set. The output port across the link is then processed identically; if both outputs want to send, the contention for the link is resolved by turning off the port which sent during the last tick. If the processed port was to the CPU, then a separate section of code is executed, to deal with the different pointers.

If the controller is idle and there is a channel waiting for it, that channel is enabled. Finally, both ports are marked as visited by "pass1". This prevents reprocessing the other link when its node is visited.

4.2.4 Generation. After all of the ports in each node have been processed, "generate" is called if the node has a message descriptor. Depending upon the kind of descriptor, a message may be generated. If one is, then a channel is created from the CPU and a pointer set in that channel to the generated message.

4.2.5 Pass two. In "pass2", the active ports have bytes moved from them. "pass2" is called from "run" if either the input, output, or CPU output port is sending. First, the output ports are processed: if the "sending channel number" flag is set, no byte is moved in order to to simulate that overhead. Otherwise, if there is room in the input buffer, a byte is moved by incrementing "inbuf" and decrementing "outbuf". If there is no room,

nothing moves and the reject flag is set. However, if the port was to the CPU, the byte will always be accepted -- this simulates a DMA channel to the X-Node's (relatively large) memory. If the byte sent was the last of the message, a trace message is output; if the message was also a teardown, the slot is removed.

The input port is processed next. If the channel is waiting for the controller, it is rejected. If it is sending to the controller, the byte is moved. If that was the last byte of the address, the message is routed and a channel is created at the selected output port. If the input port is sending to an output port, the byte is moved, subject to being rejected by the output port. The number of active channels at a port is not explicitly limited.

Next, if this is the CPU port, and the CPU is sending, a byte is moved as with the output port above. Finally, if a message is generated, a channel is created from the CPU.

4.2.6 Trace messages. The messages output by the simulator fall into three classes: run parameters, primary trace data, and secondary trace data. Generally, a message is output only if an event occurs; if a port is idle no message will be generated to describe its state. This approach reduces the amount of output done by the simulator.

Some run parameters are the system and workload variables, which are echoed by the initialization routine. Other such messages indicate the start and end of a run, the current tick, and exhaustion of free lists of messages, message descriptors, and channels. These messages are used by the statistics program to

adapt itself to the particular system being simulated.

As was discussed in the requirements, the basic performance indices are message transmission time, consecutive bytes sent, channel usage, and buffer occupancy. These data are given by the primary trace messages. The transmission time is given when each message is completely received at its destination. The number of consecutive bytes sent from each CPU is output each time a byte from the CPU is rejected. (Transmission of the channel number is not counted.) Each time a channel is added at a port, the total number of channels there is output. Buffer occupancy is not measured directly -- the frequency of rejection can indicate this. Constantly outputting the number of bytes in each buffer would be possible, but slow.

The secondary trace messages provide more detailed data about the operation of the network. This includes transmission of individual bytes, contention for links, starting transmission on a new channel, and creation of a new channel. The reject message mentioned above can be used with the new channel message to determine how many channels at a port have data to send.

## 4.3 Statistics Program

It is expected that the user of the simulator will write his own statistics-gathering program to produce the information he desires. To indicate one approach and to provide some initial data, a statistics program is provided in Appendix C. This program is designed to produce three kinds of information: average transmission time as a function of distance, channel usage sorted

by number of channels used, and consecutive bytes transmitted from CPUs, sorted by number of bytes sent. The statistics program is in Appendix C.

4.3.1 Transmission times. As was stated in the requirements, the primary performance index of the network is the time required for messages to be transmitted as a function of their size and path length. For this statistics program, all messages are assumed to be of uniform length. Each message carries with it a count of the number of communication switches it has passed through as well as the time at which it was generated. The data structure used has two counters for each path length from zero to nine ("time"). When a message arrives (indicated by an 'a' trace message), its total travel time is added to "time[i].timesum", where "i" is the path length. The number of messages having this path length ("time[i].msgnum") is then incremented. When the simulation is over, the average transmission time for each path length is computed and output.

4.3.2 Channel usage. A distribution of the numbers of active channels at each port is produced. Each time a message is routed, a 'q' trace message is output, containing the new number of active channels at the input port. The statistics routine keeps the maximum number of active channels at each port, increasing the count when a 'q' message with a greater count arrives. After the simulation ends, the array of counts ("chnls") is searched and a list made of the number of ports having each number of channels, from zero to nine ("active"). This array is then

printed out.

4.3.3 Transmission block length. Each time a message is rejected because of buffer overflow, a 'b' trace message is output. These are used to increment an array ("bytes") whose index is the number of bytes sent. This is output at the end, giving a distribution of the number of times each number of bytes was sent. This is computed only for the CPU outputs, as that is the only place 'b' messages are generated.

4.3.4 Program organization. The arrangement of the statistics program follows that of the simulator. The main routine is a while loop executed each time an 'r' message is found, indicating the start of a new run. The routine "getparams" then gets and echoes the run parameters output by "initialize" and the routine "run" reads the trace messages output by the "run" routine in the simulator and updates the data structures in the way described above. After the end of the run, the statistics are output.

"run" uses two routines: "gettrace" reads in one standard trace message without echoing it to the output. This is where the size of the output file is limited. "getnum" is identical to that in the simulator, except that it does not echo its inputs; it reads a decimal number and returns its value. The only outputs are those explicitly produced by "getparams" and "run".

The statistics program must thus accept the inputs from "initialize" as well as the standard trace messages, should echo the run parameters to the output, and must be able to take into account some of the system variables, such as the number of nodes

and ports.

# 5 TESTING

The testing of the simulator has two aspects: insuring that it is operating correctly and determining how well it operates. This must also be done when new modules are added to the program. What is required is detailed observation of runs and measurement of execution time and object code size.

## 5.1 Correctness of Operation

Verifying correct operation is done by sending messages and checking their progress and interaction at each tick. This is done by using the interactive mode and by adding diagnostic prints to supplement the standard trace messages. Testing proceeds in steps: first sending a single message along a specified path, then sending two and watching their collision, and finally using a random message generator and checking the results. At each step of this procedure, all the possible actions should be explored, that is, all paths through the simulator should be exercised.

## 5.2 Measurement

The efficiency of the simulator should be monitored to determine if the program should be modified. If it runs too slowly, unnecessary code can be removed. For example, if a series of simulations uses only one case of a particular switch

statement, the switch can be replaced by the code for only that case. If trace outputs are generated that are ignored by the statistics routine, those output statements can be removed.

Execution time should also be studied as a function of network size and traffic density. The more operations occur, the longer the simulation will require. This is a result of the event oriented approach the simulator uses.

For simulation of large networks requiring many channels, the object file may require more memory than is available on the timesharing system under which the simulator was created. If the code cannot be compacted, the simulator must be moved to another machine and possibly implemented in a different language in the process.

# CONCLUSIONS

## 6.1 Measurements

Measurements of object code size reveal one initial advantage to the use of optimization in compiling the simulator. The amount of executable code was reduced from 8602 bytes to 7818, a 9% reduction. Unfortunately, and as expected, the size of the data structures is dominant. For a tree of 128 nodes with six ports (as for a full-ring tree) and allowing 200 channels and 100 messages, the total memory required is only reduced from 30,032 bytes to 29,249. The Unix operating system allows the user only 32,768 bytes; this could be exceeded by larger trees and by more active trees (which would require more simultaneously active channels). This would mean that parts of the data structure would nave to be stored on files, greatly increasing the run time, or the simulator would have to be transported to a different host.

## 6.2 Future Directions

6.2.1 Investigations. A simulation will be run immediately to compare the run time of this simulator with that of the previous version (not implementing virtual channels). The results will be available in the Unix X-Tree library under the "simulator" directory where the program is kept. Another benchmarking study could compare the performance of the simulator with and without optimization; this is expected to be the main advantage of

optimization.

6.2.2 Simulator development. A number of improvements can be made to the simulator. Allowing multiple or more complex message descriptors at each node would permit transmitting more than one block of bytes down a virtual channel while measuring transmission times for each.

Implementation of more routing algorithms should be the next task undertaken, in particular a half-ring or fault-tolerant full-ring algorithm. The latter has been devised by Professor C. H. Sequin, but it has yet to be implemented on the simulator.

## REFERENCES

1. Despain, A. M. and D. A. Patterson. "X-Tree: a tree-structured multi-processor computer architecture." Proceedings of the Fifth Annual Symposium on Computer Architecture, 3 - 5 April 1978. Pp. 144 - 151.

2. Goodman, J. R. "A new interconnection scheme for X-Tree." Unpublished CS 292T term paper, U. C. Berkeley. 8 December 1977.

3. Tymes, L. R. "TYMNET -- A terminal oriented communication network." Proceedings of the Spring Joint Computer Conference, 1971. Pp. 211 - 216.

4. Ripley, G. D. "Program perspectives: a relational representation of measurement data." IEEE Transactions on Software Engineering, vol. SE-3, no. 4. July 1977. Pp. 296 - 300.

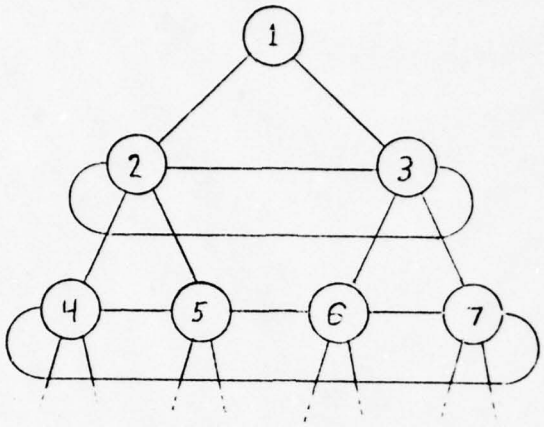



Figure 1

Full-Ring X-Tree

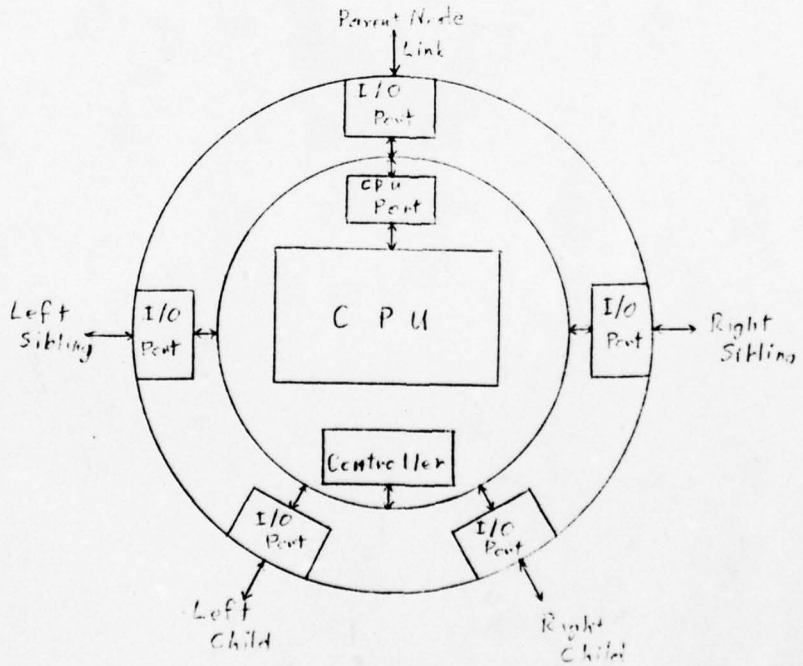



Figure 2

X-Node Organization

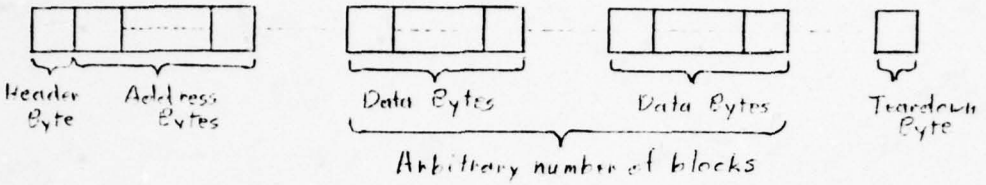



Figure 3

Message Format

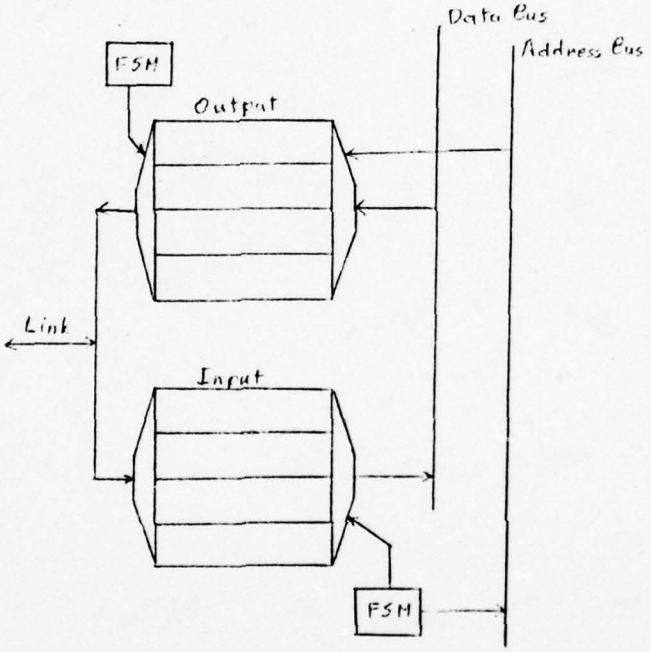



Figure 4

Port Organization

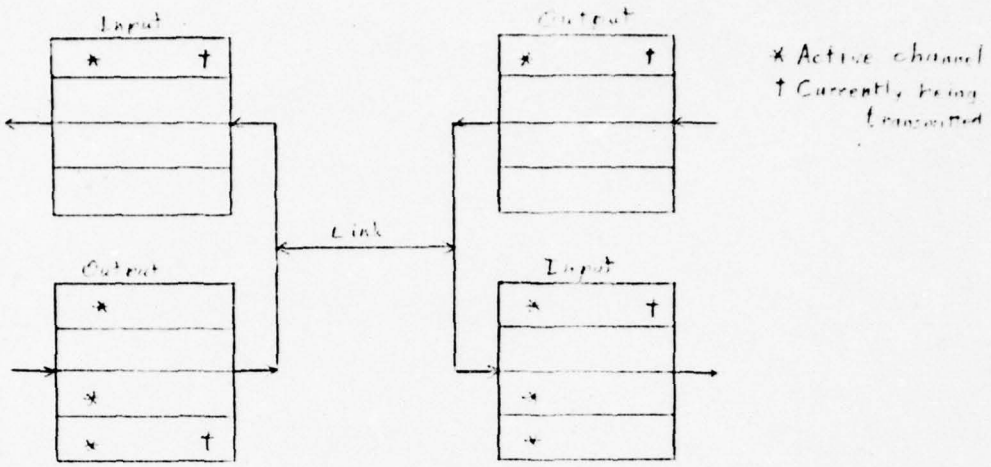



Figure 5a

Physical Channels

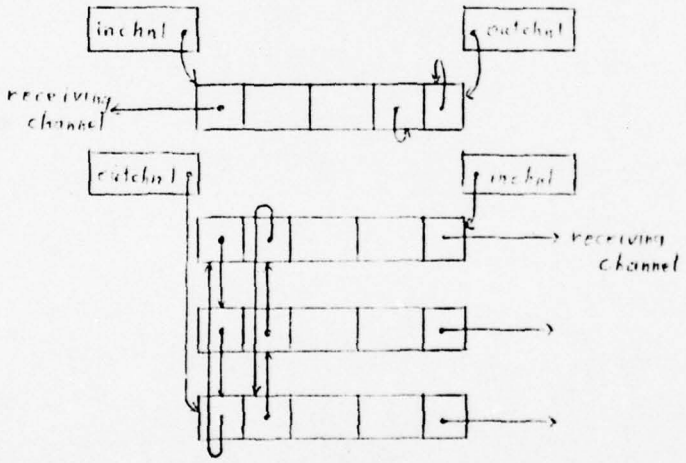
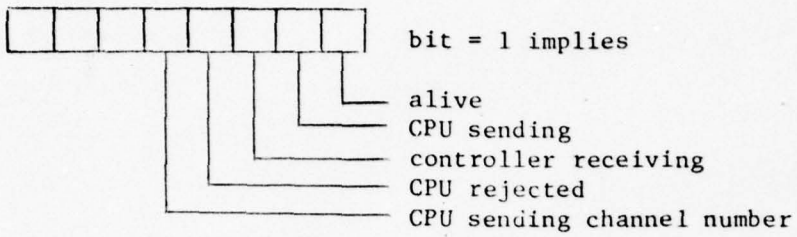


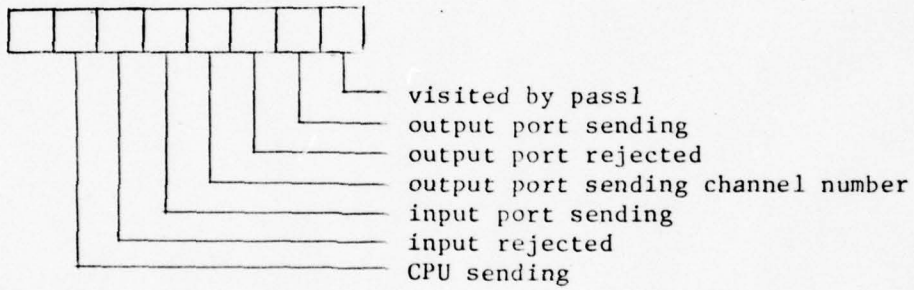

Figure 5b

Simulated Channels

Node Flag




Port Flag




Channel Flag

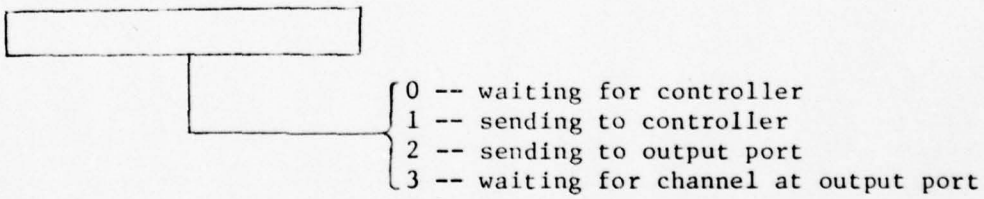



Figure 6

Flag Formats

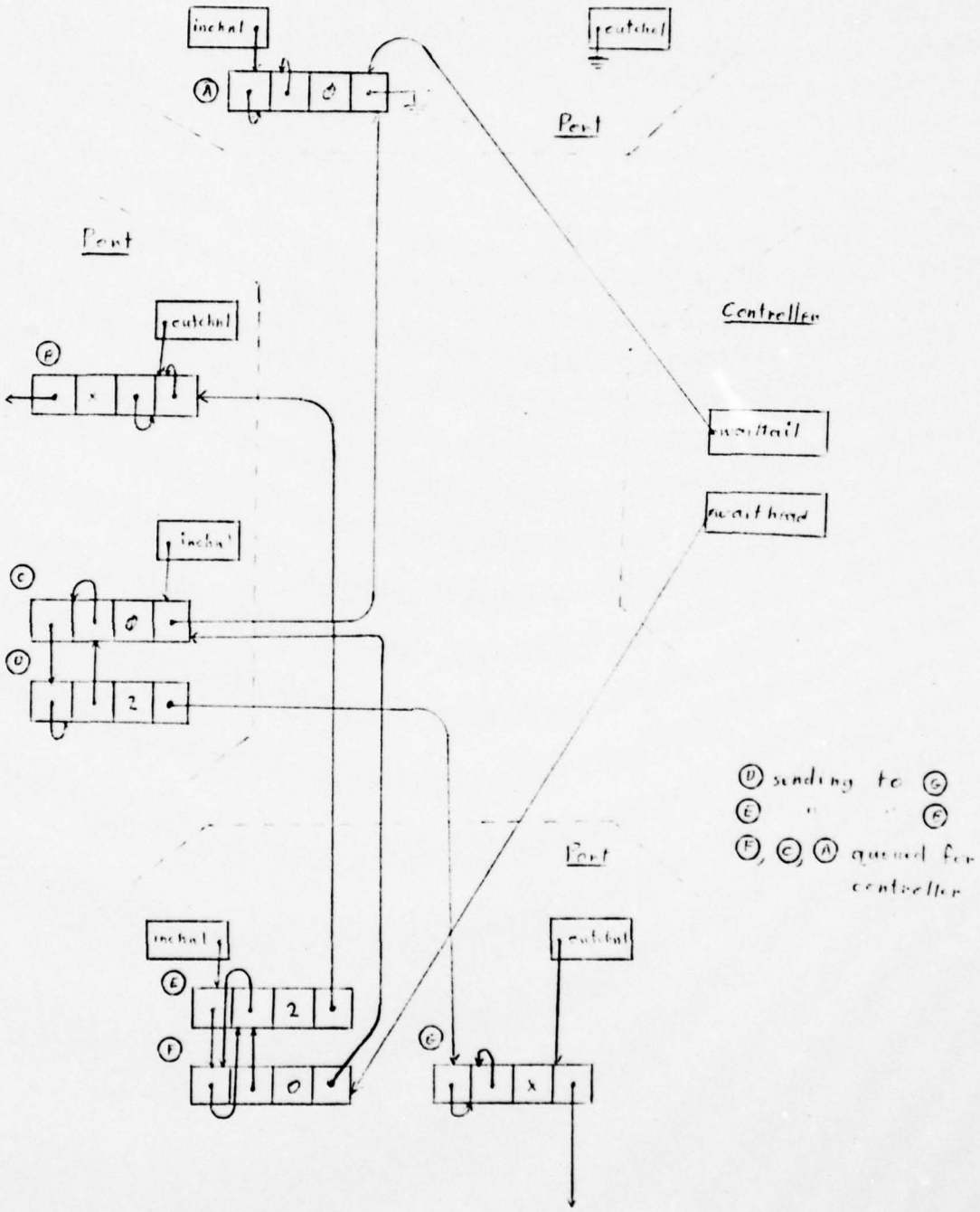



Figure 7

Switch Representation

# APPENDIX A

# XSIM USER'S GUIDE

# 1 INTRODUCTION

The X-Tree dynamic communications simulator is a program to simulate the passing of messages among the nodes in a multiprocessor computer network. While the primary application is for the hierarchical structures considered for the X-Tree system, the simulator can be used for any system using the buffering and protocols described in the Report.

The simulator accepts input from the user specifying parameter input / data output formats, workload variables, and system variables. The simulator produces as its output a trace of all activities in the network which may be piped into statistics routines for data reduction and presentation.

System variables include network structure, message routing algorithm, buffer size, and disabled links and nodes. Workload variables determine message attributes such as origin and destination, generation time (or frequency), and length.

Currently-implemented network structures include simple binary, full-ring, and half-ring. There is an option to use a binary tree with extra user-specified links. Messages can be generated either explicitly, with origin, destination, and start time given by the user, or randomly, with length, generation frequency, and transmission distance specified.

The remainder of this guide describes the the operation of the simulator from the user's point of view, its implementation, and procedures for modification.

The simulator is in the X-Tree Unix library, at

/b/xtree/library/simulator

This is a directory containing an explanatory file ("help"), the simulator ("XSIM"), the simulator source (under directory "source"), and a copy of this report ("report").

# 2 Simulator Operation

This section describes how to use the simulator. The first part discusses the various simulator, system, and workload variables, and the second part discusses the output.

## 2.1 Run Definition

Before the simulator begins execution of each run, it must be provided with information describing the system and workload to be simulated and describing the parameter input and trace output formats desired. An example run is given in section three of this guide.

2.1.1 Input formats. For user familiarization purposes, the simulator can accept run parameters in an interactive fashion, prompting the user for each item. Such a run is illustrated in Appendix D. For longer "production" runs, the parameters can be supplied from a file and the prompts suppressed. This option is selected by using the parameter "-s" when the simulator is executed. If the simulator's executable code is on file "XSIM", the shell command "XSIM" will cause execution in the interactive mode. The command "XSIM -s" will cause execution without prompts. To supply parameters from file 'PARAMS', the command would be

"XSIM -s <PARAMS"

2.1.2 Output formats. As mentioned before, the simulator produces a trace of all activities in the network. For interactive runs, these trace messages can be given in clear english, while for runs whose data is piped through a statistics program the messages can be given in an abbreviated, easy-to-digest form.

This selection is made with the first parameter: 1 selects the english output and 2 selects the appreviated output. These are further discussed in section 2.2.

To use the simulator with a statistics program "STATS" and store the data on the file "DATA", the command would be

"XSIM -s <PARAMS | STATS >>DATA"

The double right arrow will catenate the output data onto the end of the data file, thus allowing results of several runs to be put onto the same file. This also gives a mechanism to record the execution time of each run, by using the Unix 'date' command, which gives the date and time. Two runs with parameters on separate files could be placed in a shell program:

```
date >>DATA
XSIM -s <PARAM1 | STAT >>DATA
date >>DATA
XSIM -s <PARAM2 | STAT >>DATA
date >>DATA
```

2.1.3 System definition. The output format is the first parameter input; it causes outputs to be either English messages or abbreviated messages consisting of a character and three integers (these will be explained below). The next group of parameters describe the system being simulated. These choose the buffer sizes, number of nodes, the network structure, dead links and nodes.

The debugging level can be set to one to cause a dump of the state of the network after each pass through the tree to find sending nodes. The dump shows pointers, flags, and the contents of channels, messages, and message descriptors.

The input and output buffer sizes can each be set independently to one to 127 bytes.

The parameter for the number of nodes is a positive integer less than or equal to 127. If more nodes are desired, the source code must be modified and recompiled.

There are four choices for the structure, indicated by the integers one to four. One selects the simple binary tree, where port 0 goes to the parent, 1 to the left child, and 2 to the right child. Two selects the full-ring tree, where ports 0 to 2 are as before, 3 goes to the left sibling, and 4 to the right sibling. Structure three selects the half-ring tree, with port 2 being the connection to sibling.

Four selects a binary tree with one link to other siblings. Following the structure parameter, the user enters one number for each node, indicating the node to which the fourth port is attached. If two conflicting link definitions are given, the second will be used.

For structure four the user must supply a routing algorithm to make use of the interconnections. This involves adding a case to the switch statement in the ROUTE subroutine. In addition, there is no routing algorithm installed for structure three (half-ring). Because of this, the initialization routine will not accept structures three and four.

The user can choose to disable selected links or nodes in

the structure. Disabling a node or link is done by entering a line of the form:

-1 2 3.1.2 4.3.1 5

The given line will disable nodes 1, 2, and 5, and ports 1 and 2 of node 3 and ports 1 and 3 of node 4. The numbers are decimal digits -- no non-digits except '-' appear in the command string.

The final system variable is the routing algorithm. One is the simple binary tree algorithm, two is the simple full-ring algorithm.

2.1.4 Workload definition. The workload is embodied in the messages sent through the network. The user can define message descriptors to generate different kinds of messages. There are four types of descriptors currently implemented: specified messages, random messages to non-leaves, leaf-to-leaf messages with flat distribution, and leaf-to-leaf with normal distribution.

For each descriptor of any type, the user specifies the address and total length, in bytes. For type one (specified) descriptors, the start time and destination node are also specified. For types two, three, and four, the frequency of generation is specified as the ratio of two integers. For type two descriptors, the destination is chosen at random from the non-leaves; for type three, from the leaves, with each leaf having equal probability. For type four descriptors, a standard deviation is also given, with its units representing the horizontal distance in nodes from the sender. This descriptor thereby sends from leaf-to-leaf with the nearer leaves selected more often to

simulate communication locality. Finally, the kind of the message is set to either teardown or non-teardown.

Several different descriptors of each type can be defined, each having a different set of nodes using it. For example, some leaves can generate messages to other leaves with one standard deviation and other leaves can have another. If a node is assigned no descriptor, it will generate no messages.

The user may add his own workload-producing routine, but this will require modification of the initialization and generation routines.

2.1.5 Other parameters. After specifying the workload, the user must give the length of the run in ticks. This is a positive integer.

After the run is finished, the simulator reads the input to determine whether to start on another run. The letter "y" will cause the simulator to reinitialize and execute, any other will cause program termination. Obviously, if another run is started, the parameter file must contain another set of parameters.

2.1.6 Parameter retention. Most parameters are kept the same from run to run. By entering a zero or a return the old parameter will be used. The two exceptions are for dead links and message descriptors; both of these are cleared and must be reentered with each run.

2.2 Trace Output

The simulator produces a series of messages describing the progress of the simulation. For output format one (english) the messages are self-explanatory. For format two (abbreviated), they are of the form <letter> <integer1> <integer2> <integer2>.

Five of the twelve messages provide information about the simulator. By letter, they are:

r: Start of a new run. The three integers are zero. The following integers are the run parameters being echoed.

t: Start of a new tick. <integer1> is the tick number, and the other two fields are zero.

n: No more items from a free list. This indicates that either messages, descriptors, or channels are being used faster than they are being released and the appropriate free list has run dry. <integer1> is the node number, <integer2> is the port, and <integer3> is 0 for messages, 1 for channels, or 2 for descriptors.

e: End of run. Other fields are zero. The next character will be the echo of the continuation ('y' or 'n').

x: End of session. Other fields are zero. This is the last message.

The other eight messages describe the progress of messages through the system.These are:

a: Message has arrived at its destination. (Last byte sent to the CPU.) <integer1> is the destination node, <integer2> is the path length, and <integer3> is the start time.

b: Transmission has been interrupted after a string of several consecutive bytes. <integer1> is the sending node,

<integer2> the port, and <integer3> the number of bytes. These messages are currently only generated for the CPU output.

f: Buffer full -- a byte was rejected because the receiving buffer was full. <integer1> is the sending node, <integer2> the output port, and <integer3> is 0 if an input port, 1 if an output port, 2 if an input sending to the controller, and 3 if an output from the CPU.

g: Get new channel -- a new channel has been selected for transmission. This message is suppressed for input ports when a new channel is selected after every byte is sent. <integer1> is the node, <integer2> is the port>, and <integer3> is 0 for an input port and 1 for a output port.

l: Both nodes on a link could send and only one did. <integer1> is a node, <integer2> is its port, and <integer3> is 0 if that node sent, 2 if the other sent.

q: An address was received, the message routed, and a channel created at the selected output port. <integer1> is the node, <integer2> the input port, and <integer3> is the number of channels at the input port.

s: One byte sent at a port. <integer1> is the node, <integer2> is the port, and <integer3> is 0 for an input port sending to an output, 1 for an output sending across a link, 2 for the CPU output, and 3 for an input sending to the controller.

## 3 EXAMPLE INTERACTIVE RUN

Bracketed statements are comments.

```
What output format:
    1:  English
    2:  Abbreviated
 Currently 1:1
What debugging level:
    1:  none
    2:  dump after pass1
 Currently 1 :1
What size input buffer? (0):4
What size output buffer? (0):4
 Does input port send more than one consecutive byte per channel?
      1: No
      2: Yes
  Currently: 1 :2
How many nodes in tree? (0):3
What tree structure:
    1:  Simple binary
    2:  Full-ring
    3:  Half-ring*
    4:  Binary with one user-defined link*
    *:  No routing algorithms
Currently 1:1
Kill nodes and links
 '-' to kill, return to exit
  Specify as '-n n n.p n.p.p', etc.
                                        [return entered to exit]

What routing algorithm?
    1: Simple binary
    2: Full-ring
 Currently 1: 1
Set up message descriptors and assign nodes to them
Enter descriptor type:
    1: Specified
    2: Random to non-leaf
    3: Leaf-to-leaf with flat distribution
    4: Leaf-to-leaf with normal distribution
    Enter return to quit
1
 Enter address length and total length:
1 4
 Enter start time and destination:
1 2
 Is this a teardown (0 = no, 1 = yes):0
 Enter nodes to use this descriptor
1
Enter descriptor type:
    1: Specified
```

```
    2: Random to non-leaf
    3: Leaf-to-leaf with flat distribution
    4: Leaf-to-leaf with normal distribution
    Enter return to quit                          [return entered to exit]
What length of run in ticks? (0):15
Start of run at node 100, port 4: 0               [node and port meaningless]
time is 1                                         [message generated]
time is 2
Start new channel at node 1, port 3: 0            [only channel number sent]
time is 3
Start new channel at node 1, port 3: 0
One byte sent at node 1, port 3: 2                [#1 from CPU to port 3 input]
time is 4
One byte sent at node 1, port 3: 3                [address byte to controller]
Channel created at node 1, port 3: 0              [at output port to node 2]
One byte sent at node 1, port 3: 2                [#2 from CPU to input]
time is 5
Start new channel at node 1, port 1: 0
Start new channel at node 1, port 1: 1
One byte sent at node 1, port 3: 0                [#1 from input from CPU to output]
One byte sent at node 1, port 3: 2                [#3 from CPU]
time is 6
Start new channel at node 1, port 1: 0
One byte sent at node 1, port 3: 0                [#2 from CPU input to output]
One byte sent at node 1, port 3: 2                [#4 from CPU]
                                                  [channel number across link]
time is 7
Start new channel at node 1, port 1: 0
Start new channel at node 1, port 3: 1
One byte sent at node 1, port 1: 1                [#1 across link]
One byte sent at node 1, port 3: 0                [#3 from CPU input]
time is 8
Start new channel at node 1, port 3: 1
One byte sent at node 1, port 1: 1                [#2 across link]
One byte sent at node 1, port 3: 0                [#4 from CPU input]
One byte sent at node 2, port 0: 3                [address byte to node 2 controller]
Channel created at node 2, port 0: 0              [to node 2 CPU]
time is 9
Start new channel at node 1, port 3: 0
Start new channel at node 1, port 3: 1
Start new channel at node 2, port 3: 1
One byte sent at node 1, port 1: 1                [#3 across link]
One byte sent at node 2, port 0: 0                [#1 from input from link]
time is 10
Start new channel at node 1, port 3: 0
Start new channel at node 1, port 3: 1
One byte sent at node 1, port 1: 1                [#3 across link]
One byte sent at node 2, port 0: 0                [#2 from input from link]
time is 11
Start new channel at node 1, port 1: 1
Start new channel at node 1, port 3: 0
Start new channel at node 1, port 3: 1
One byte sent at node 2, port 0: 0                [#3 from input from link]
One byte sent at node 2, port 3: 1                [#1 to CPU]
time is 12
Start new channel at node 1, port 1: 1
```

```
Start new channel at node 1, port 3: 0
Start new channel at node 1, port 3: 1
One byte sent at node 2, port 0: 0       [#4 from input from link]
One byte sent at node 2, port 3: 1       [#2 to CPU]
time is 13
Start new channel at node 1, port 1: 0
Start new channel at node 1, port 1: 1
Start new channel at node 1, port 3: 0
Start new channel at node 1, port 3: 1
One byte sent at node 2, port 3: 1       [#3 to CPU]
time is 14
Start new channel at node 1, port 1: 0
Start new channel at node 1, port 1: 1
Start new channel at node 1, port 3: 0
Start new channel at node 1, port 3: 1
One byte sent at node 2, port 3: 1       [#4 to CPU]
Message arrives at 2 from 1 since 1
time is 15
Start new channel at node 1, port 1: 0   [search in vain for channel
Start new channel at node 1, port 1: 1    with data]
Start new channel at node 1, port 3: 0
Start new channel at node 1, port 3: 1
Start new channel at node 2, port 3: 1
End of run at node 4, port 4: 0          [node and port meaningless]

Do you wish to continue (y-n)?:n
End of session at node 4, port 4: 0      [node and port meaningless]
```

# APPENDIX B

## SIMULATOR CODE

```
In the "printf" statements in this program and in the statistics
routine in Appendix C, the 'backslash-n', or end-of-line, is
printed as a pound sign ('#').
#
/* X-Tree Dynamic Communications Simulator

Paul A. Suhler
14 May 1978


Constant Declarations */

#define NODEMAX 128     /* nodes in tree */
#define PORTMAX 6       /*ports per node, incl. cpu */
#define CHNLMAX 200     /* chnls in system */
#define MSGMAX  100     /* messages in system */
#define DESCMAX 5       /* different message descriptors */

/* Type Declarations */


struct msdstype {struct msdstype *nextd;
                 char dkind, aleng, mkind; int mleng, i1, i2;
                 float freq;
                };
struct msgtype  {struct msgtype *nextm; int source, dest, stime, length;
                 char kind, adleng;
                };
struct chnltype {struct chnltype *nextc, *lastc, *destc;
                 struct msgtype  *cmsg; char outbuf, inbuf, cflag;
                };
struct porttype {struct chnltype *inchnl, *outchnl;
                 struct nodetype *prnode; struct msgtype *qhead, *qtail;
                 struct porttype *link;
                 int chnls; char pflag;
                };
struct nodetype {struct porttype port[PORTMAX]; char nflag, cpchnls, conbuf;
                 struct chnltype *cpoutchnl, *waithead, *waittail;
                 struct desctype *msgdesc;
                 int cpbytes;
                };

/* Variable Declarations */

struct msdstype descriptor[DESCMAX], *freedesc;
struct chnltype chnl[CHNLMAX], *freechnl;
struct msgtype  message[MSGMAX], *freemsg;
struct nodetype node[NODEMAX];

int  nodenum, portnum, chnlnum, inbuflim, outbuflim;
char shape, rtalgo, prompt, output, c, getnew, debug;

int n, p, tick, timelim;
```

```
#include "s4h"

/* MAIN -- Upon initial entry, the prompt option is cleared if the
simulator was called with a '-s' argument.  Then, each time a
simulaton is run, the simulator is initialized and the run is made.
After the run, the simulator terminates if a 'y' is not read in. */

main(argc,argv)
  int argc; char **argv;
  {char c;
   prompt =! ((argc > 1) && (argv[1][1] == 's'));
mlp:    initialize();
        run();
        if (prompt) printf("##Do you wish to continue (y-n)?:");
        c = putchar(getchar());  getchar();
        if (c == 'y')  goto mlp;
        else
          printmsg('x',0);
  };    /* main */


/* RUN -- run clears certain pointers and flags, sets up free lists
of messages and channels, and then runs the simulation according to the
parameters read in in "initialize".  For the first pass, "pass1" is called
if the node is alive and the port has a live link and is unvisited.
After each node is processed, "generate" is called if there is a message
descriptor at that node.  For the second pass, if the node is alive and
the port is marked as sending, "pass2" is called. */

run()
  { register struct porttype *tport;
    register struct nodetype *tnode;
    /* clear nodes and ports */

    for (n=1; n<=nodenum; n++)
    { tnode = &(node[n]);
      tnode->cpchnls = tnode->conbuf = tnode->cpbytes = 0;
      tnode->cpoutchnl = tnode->waithead = tnode->waittail = 0;
      for (p=0; p<=portnum; p++)
      { tport = &(tnode->port[p]);
        tport->inchnl = tport->outchnl = tport->qhead = 0;
        tport->chnls = tport->pflag = 0;
        tport->prnode = tnode;
      }
    }

    /* set up lists of free messages and chnls */

    for (n=0; n<CHNLMAX; n++)
      chnl[n].nextc = &(chnl[n+1]);
    chnl[CHNLMAX - 1].nextc = 0;
    freechnl = &(chnl[0]);

    for (n=0; n<MSGMAX; n++)
      message[n].nextm = &(message[n+1]);
    message[MSGMAX - 1].nextm = 0;
```

```
    freemsg = &(message[0]);

    /* run simulation */

    printmsg('r',0);

    for (tick=1; tick<=timelim; tick++)
    { printmsg('t', tick);
      for (n=1; n<=nodenum; n++)
        if ((tnode = &(node[n]))->nflag & 0001)
        { /* node is alive */
          for (p=0; p<=portnum; p++)
          { if ((((tport = &(tnode->port[p]))->pflag & 1) == 0) &&
                (tport->link))
              /* port unvisited this tick and link alive */
              pass1(tport);
          }
          if (tnode->msgdesc)
            generate(tport, tnode->msgdesc);
        }

      if (debug == 2)
        dump();

      for (n=1; n<=nodenum; n++)
        if ((tnode = &(node[n]))->nflag & 0001)
          /* if node is alive, process */
          for (p=0; p<=portnum; p++)
          { if ((tport = &(tnode->port[p]))->pflag & 0122)
              /* input, output, or cpu port sending */
              pass2(tport);
            tport->pflag =& 0376;       /* mark unvisited */
          }
      if (debug == 2)
        dump();
    }
    printmsg('e',0);
}     /* run */
```

```
#include "s4h"

/* INITIALIZE */

initialize()
  { register int num;
    int m;
    register struct msdstype *pdesc;
    register struct porttype *pp;

    /* get output format */
    if (output == 0)
      output = 1;
    if (prompt)
    { printf("What output format:# );
      printf("     1:  English# );
      printf("     2:  Abbreviated# );
      printf(" Currently %d:",output);
    }
    output = ((num = getnum()) ? num : output);
    printf("%d# ",output);

    /* get debugging level */

    if (debug == 0)
      (debug = 1);
    if (prompt)
    { printf("What debugging level:# );
      printf("     1:  none# );
      printf("     2:  dump after pass1#");
      printf(" Currently %d :",debug);
    }
    debug = ((num = getnum()) ? num : debug);
    printf("%d# , debug);

    /* get input buffer limit */

ibuflp:
    if (prompt)
      printf("What size input buffer? (%d):",inbuflim);
    if (((num = getnum()) < 0) || (num > 127))
    { printf(" Wrong -- must be in range 1 to 127 bytes. Try again. #");
      goto ibuflp;
    }
    else
      inbuflim = (num ? num : inbuflim);
    printf("%d# ",inbuflim);

    /* get output buffer limit */

obuflp:
    if (prompt)
      printf("What size output buffer? (%d):",outbuflim);
    if (((num = getnum()) < 0) || (num > 127))
    { printf(" Wrong -- must be in range 1 to 127. Try again. #");
      goto obuflp;
```

```
    }
    else
      outbuflim = (num ? num : outbuflim);
    printf("%d#",outbuflim);

    if (getnew == 0)
      getnew = 1;
    if (prompt)
    { printf(" Does input port send more than one consecutive byte per");
      printf(" channel?#      1: No#      2: Yes#  Currently: %d :",getnew);
    }
    getnew = ((num = getnum()) ? num : getnew);
    printf("%d#",getnew);

    /* get number of nodes in tree */

nnumlp:
    if (prompt)
      printf("How many nodes in tree? (%d):",nodenum);
    if ((num = getnum()) >= NODEMAX)
    { printf(" Sorry, maximum of %d nodes allowed. #",NODEMAX - 1);
      printf(" to get more, modify source and recompile. #");
      goto nnumlp;
    }
    else
      nodenum = (num ? num : nodenum);
    printf("%d#", nodenum);

    /* get structure */

    if (shape == 0)
      shape = 1;
shaplp:
    if (prompt)
    { printf("What tree structure:#");
      printf("    1:  Simple binary#");
      printf("    2:  Full-ring#");
      printf("    3:  Half-ring*#");
      printf("    4:  Binary with one user-defined link*#");
      printf("    *:  No routing algorithms# );
      printf("Currently %d:",shape);
    }
    if (c = getnum())
    { if ((c >= 1) && (c <= 2))
        shape = c;
      else
      { printf(" Invalid selection; try again;#");
        goto shaplp;
      }
    }
    printf("%d#",shape);
    /* set portnum */
    switch(shape)
    { case 1: portnum = 3; break;
      case 2: portnum = 5; break;
      case 3: case 4: portnum = 4; break;
```

```
    }

    /* initialize entire structure */

    for (n=1; n<=nodenum; n++)
    { for (p=0; p<portnum; p++)
      { if((pp = &(node[n].port[p]))->link = otherport())
          pp->link->link = pp;
      }
      node[n].port[portnum].link = 1;
      node[n].msgdesc = 0;
      node[n].nflag = 1;
    }

    /* kill nodes and ports */

    if (prompt)
    { printf("Kill nodes and links#");
      printf(" '-' to kill, return to exit#");
      printf("  Specify as '-n n n.p n.p.p', etc.##");
    }
    while ((c = putchar(getchar())) == '-')
      while (c != '#')
      { n = getnum();
        printf("%d%c",n,c);
        if ((n >= 1) && (n <= nodenum))
        { if ((c == ' ') || (c == '#'))
          { node[n].nflag =& 0376;
            for (p=0; p<portnum; p++)
              if (pp = otherport())
                pp->link = 0;
          }
          else
            while (c == '.')
            { p = getnum();
              printf("%d%c",p,c);
              if ((p >= 0) && (p <= portnum))
              { pp = &(node[n].port[p]);
                { if (pp->link > 1)
                    pp->link->link = 0;
                  pp->link = 0;
                }
              }
              else
                printf("Port number %d invalid#",p);
            }
        }
        else
          printf("Node number %d invalid#",n);
      }

/* Select routing algorithm */

    if (rtalgo == 0)
      rtalgo = 1;
```

```
rtlp:
    if (prompt)
    { printf("What routing algorithm?#");
      printf("     1: Simple binary#");
      printf("     2: Full-ring#");
      printf(" Currently %d: ",rtalgo);
    }
    if (((num = getnum()) >= 0) || (num <= 2))
      rtalgo = (num ? num : rtalgo);
    else
    { printf(" Invalid selection; try again#");
      goto rtlp;
    }
    printf("%d#", rtalgo);

/* set up message descriptors */

    for (n=0; n<DESCMAX-1; n++)
      descriptor[n].nextd = &(descriptor[n+1]);
    descriptor[DESCMAX-1].nextd = 0;
    freedesc = &(descriptor[0]);

    if (prompt)
      printf("Set up message descriptors and assign nodes to them#");
    while (n)
    { if (prompt)
      { printf("Enter descriptor type:#");
        printf("     1: Specified#");
        printf("     2: Random to non-leaf#");
        printf("     3: Leaf-to-leaf with flat distribution#");
        printf("     4: Leaf-to-leaf with normal distribution#");
        printf("     Enter return to quit#");
      }
      if (((n = getnum()) >= 1) && (n <= 4))
      { printf("%d#", n);
        if (pdesc = freedesc)
        { freedesc = pdesc->nextd;
          pdesc->dkind = n;
          if (prompt)
            printf(" Enter address length and total length:#");
          pdesc->aleng   = getnum(); printf("%d ",pdesc->aleng);
          pdesc->mleng   = getnum(); printf("%d#",pdesc->mleng);
          switch(n)
          {   case 1:   /* specified */
                        if (prompt)
                          printf(" Enter start time and destination:#");
                        pdesc->i1 = getnum(); printf("%d ",pdesc->i1);
                        pdesc->i2 = getnum(); printf("%d#",pdesc->i2);
                        break;
              case 4:   /* normal leaf-leaf */
                        if (prompt)
                          printf(" Enter standard deviation:#");
                        pdesc->i1 = getnum(); printf("%d#",pdesc->i1);
              case 2:   /* page */
              case 3:   /* leaf-leaf, flat */
                        if (prompt)
```

```
                    printf(' Enter frequency as 'j/k':");
                  num = getnum();
                  pdesc->freq = num;
                  printf("%d/",num);
                  if (m = getnum())
                    pdesc->freq =/ m; printf('%d#",m);
                  break;
          }     /*   switch */

          if (prompt)
            printf(" Is this a teardown (0 = no, 1 = yes):");
          pdesc->mkind = getnum();
          printf("%d#",pdesc->mkind);

          if (prompt)
            printf(" Enter nodes to use this descriptor#");
          c   = 'a';
          while (c != '#')
          { if (((num = getnum()) >= 1) && (num <= nodenum))
            { node[num].msgdesc = pdesc; printf("%d ",num); }
            else
              printf(" Invalid node number:%d#",num);
          }
          printf("#");
        }
        else
          printmsg('n',2);
      }
      else
        if (n)
          printf(" Invalid selection; try again. #");
    }

    /* get length of run */

    if (prompt)
      printf("What length of run in ticks? (%d):", timelim);
    if (num = getnum())
      timelim = (num ? num : timelim);
    printf("%d#",timelim);

  }     /* initialize */
```

```
#include "s4h"

/* PASS1 -- pass one is called by run for those ports with non-zero
links (not already visited) in alive nodes.  Pass one marks those ports
having channels with data to send.  This is done for the input and output
at both the current port and for the input and output at the port indicated
by "link".
        For input ports, the sending flag is set immediately if there is
more data in "inbuf", if there was no reject, and if "getnew" is set to 2.  If
any of these fails, findinchnl is used to search for the next channel with
data; if found, the "inchnl" pointer is reset and the sending flag is set.
If there are no channels with data, the sending flag is cleared.
        For output ports, a similar procedure is followed, except that when
a new channel is selected, the sending channel number flag is also set.  The
output to CPU is processed in a separate section of code because the CPU's
representation is actually a part of the current node.  Before the output
sending flags are set, that in the current port is saved.  If both ports
want to send, the contention can then be resolved in favor of the port
that did not send in the last tick.
*/

pass1(port)
  register struct porttype *port;
  { register struct chnltype *chnl;
    struct chnltype *nchnl;
    register struct porttype *oport;
    struct nodetype *pnode;
    char flag;

    /* This input port */

    if (chnl = port->inchnl)
    { /* there is an active channel */
      if ((getnew == 2) && (chnl->inbuf) && ((port->pflag & 0040) == 0))
        /* more data and not rejected */
        port->pflag =| 0020;
      else      /* either current channel rejected or no data -- find new */
      { if (nchnl = findinchnl(chnl))
        { /* there is a channel with data */
          port->inchnl = nchnl;
          port->pflag =| 0020;
        }
        else    /* no channels with data */
          port->pflag =& 0357;
      }
      port->pflag =& 0337;      /* clear reject */
    }
    else        /* no active channels -- mark 'not sending' */
      port->pflag =& 0357;

    /* Other input port */

    if (p < portnum)    /* check not CPU -- no other port */
    { if (chnl = (oport = port->link)->inchnl)
      { /* link is there and there is an active channel */
        if ((getnew == 2) && (chnl->inbuf) && ((oport->pflag & 0040) == 0))
```

```
        /* more data and not rejected */
        oport->pflag =| 0020;
      else     /* either current channel rejected or no data -- find new */
      { if (nchnl = findinchnl(chnl))
        { /* there is a channel with data */
          oport->inchnl = nchnl;
          oport->pflag =| 0020;
        }
        else  /* no channels with data */
          oport->pflag =& 0357;
      }
      oport->pflag =& 0337;   /* clear reject */
    }
    else      /* no active channels -- mark "not sending" */
      oport->pflag =& 0357;
}

/* determine if this output port wants to send */

flag = port->pflag & 0002;  /* save whether it was sending */
if (chnl = port->outchnl)
{ /* if any active channels */
  if ((chnl->outbuf) && ((port->pflag & 0004) == 0))
    /* data in current channel and not rejected */
    port->pflag =| 0002;
  else
  { if (nchnl = findoutchnl(chnl))
    { /* there is a channel with data */
      port->outchnl = nchnl;
      port->pflag =| 0012;  /* sending channel number */
    }
    else     /* no channels with data */
      port->pflag =& 0375;
  }
  port->pflag =& 0373;      /* clear reject */
}
else         /* no channels -- not sending */
  port->pflag =& 0375;

/* determine if other port on link wants to send and then resolve
   contention for link.  done separately for cpu and non-cpu links */

if (p < portnum)
{ /* link to other node */
  if (chnl = (oport = port->link)->outchnl)
  { /* active channel there */
    if ((chnl->outbuf) && ((oport->pflag & 0004) == 0))
      /* current channel has data and was not rejected */
      oport->pflag =| 0002;
    else
    { if (nchnl = findoutchnl(chnl))
      { /* channel exists with data */
        oport->outchnl = nchnl;
        oport->pflag =| 0012;
      }
      else  /* no channel with data */
```

```
            oport->pflag =& 0375;
        }
        oport->pflag =& 0373;   /* clear reject */
      }
      else      /* no channels */
        oport->pflag =& 0375;

      /* if both want to send, give link to one that didn't send last */

      if ((port->pflag & 0002) && (oport->pflag & 0002))
      { if (flag)
          port->pflag =& 0375;
        else
          oport->pflag =& 0375;
        printmsg('l',flag);
      }

    }   /* link to other node */
    else
    { /* link to cpu */
      pnode = port->prnode;
      if (chnl = pnode->cpoutchnl)
      { /* channel is active */
        if ((chnl->outbuf) && ((pnode->nflag & 0010) == 0))
          /* current channel has data and not rejected */
          port->pflag =| 0100;
        else    /* current channel either rejected or no data */
        { if (nchnl = findoutchnl(chnl))
          { /* there exists channel with data */
            pnode->cpoutchnl = nchnl;
            pnode->nflag =| 0020;       /* sending channel address */
            port->pflag =| 0100;
          }
          else  /* no active channels */
            port->pflag =& 0277;
        }
        pnode->nflag =& 0367;   /* clear reject */
      }
      else      /* no active channel */
        port->pflag =& 0277;

      /* if contention, resolve as before */

      if ((port->pflag & 0002) && (port->pflag & 0100))
      { if (flag)
          port->pflag =& 0375;
        else
          port->pflag =& 0277;
      }


      /* start new message to controller, if any */

      if ((chnl = pnode->waithead) && ((pnode->nflag & 0004) == 0))
      { pnode->nflag =| 0004;
        pnode->waithead = chnl->destc;
```

```
            chnl->cflag = 1;
          }

        }   /* cpu link */

        /* mark port as visited */
        port->pflag =| 0001;
        if (p < portnum)
          oport->pflag =| 0001;

      }     /* pass 1 */

/* FINDINCHNL -- this is passed a non-zero pointer to a channel in a list at a
port (a channel that was rejected on the last transmission attempt).  It
searches for the next having <inbuf> not zero and returns a pointer to
that channel.
If no channel with data is found, it will stop at the original channel,
returning a pointer to it if it has data, and zero if it has none. */

findinchnl(oc)
  register struct chnltype *oc;
  { register struct chnltype *cc;

    if (getnew == 2)
      printmsg('g',0);
    cc = oc->nextc;
    while (cc->inbuf == 0)
    { cc = cc->nextc;
        if (cc == oc)   /* if none with data */
        return ((oc->inbuf) ? oc : 0);
    } /* while */
    return(cc);
  }  /* findinchnl */


/* FINDOUTCHNL -- this routine functions identically to findinchnl, except
that it tests <outbuf>. */

findoutchnl(oc)
  register struct chnltype *oc;
  { register struct chnltype *cc;

    printmsg('g',1);
    cc = oc->nextc;     /* search for channel with data */
    while (cc->outbuf == 0)
    { cc = cc->nextc;
      if (cc == oc)     /* if no others with data */
        return ((oc->outbuf) ? oc : 0);
    }
    return(cc);
  }  /* findoutchnl */
```

```
#include "s4h"

/* GENERATE -- generate takes a pointer to a port and a copy of the message
descriptor pointer from that port and may generate a message, depending upon
the descriptor, which may involve a random function.  If a message is
produced, a channel from the CPU is made for the message, and a pointer to the
channel is returned by generate. */

generate(pport, desc)
  struct porttype *pport;
  register struct msdstype *desc;

  { struct chnlype *chnl;
    register struct msgtype *msg;
    register int leafnum;
    int destination, offset;
    extern float normal();

    destination = 0;
    switch(desc->dkind)
    { case 1:   /* specified message */
        if (desc->i1 == tick)
          destination = desc->i2;
        break;
      case 2:   /* page to non-leaf */
        if (desc->freq >= (rand() / 32767.0))
          destination = (rand() % (nodenum / 2)) + 1;
        break;
      case 3:   /* leaf-to-leaf with flat distribution */
        if (desc->freq >= (rand() / 32767.0))
        { leafnum = (nodenum + 1) / 2;
          offset = (rand() % (leafnum - 1)) + 1;
          destination = n + offset;
          if (destination > nodenum)
            destination =- leafnum;
          else
            if (destination < leafnum)
              destination =+ leafnum;
        }
        break;
      case 4:  /* leaf-to-leaf with normal distribution */
        if (desc->freq >= (rand() /32767.0))
        { leafnum = (nodenum+ 1) / 2;
          while (((offset = (desc->i1) * normal()) > (leafnum / 2)) ||
                 (offset < -(leafnum / 2)) ||
                 (offset == 0))
          ;
          destination = n + offset;
        }
        break;
    }   /* switch */

    if (destination)     /* if message generated */
    { if (msg = freemsg)
      { freemsg = msg->nextm;
        msg->source = n;
```

```
            msg->dest = destination;
            msg->kind = desc->mkind;
            msg->stime = tick;
            msg->length = desc->mleng;
            msg->adleng = desc->aleng;
            return(mkchnl(pport, 0, msg));
          }
          else
          { printmsg('n',0);
            return(0);
          }
        }
      }     /* generate */

/* ROUTE -- route takes a pointer to a message and uses the global node 'n'
to route the message according to the algorithm specified by 'rtalgo'. */

struct chnltype *route(msg)
  struct msgtype *msg;
  { struct porttype *port;
    register int target, addr, oldaddr;
    int value, dist;

    addr = n;
    target = msg->dest;
    switch(rtalgo)
    { case 1:   /* simple binary tree */
        if (target == addr)
         value = portnum;
        else
        { while (target > addr)
          { oldaddr = target;
            target = target / 2;
          }
          if (target < addr)
            value = 0;
          else
            value = ((oldaddr & 0001) ? 2 : 1);
        }
        break;
      case 2:  /* full-ring (not fault-tolerant) */
        if(target == addr)
           value = 5;
        else
           { if (higher(target, addr))
                value = 1;
             else
                { while (higher(addr, target))
                  { oldaddr = target;
                    target =/ 2;
                  }
                  if (target == addr)
                     value = ((oldaddr & 1) ? 2 : 1);
                  else
                     { dist = mindist(target, addr);
```

```
                        if (((dist > 0) ? dist : -dist) >= 5)
                           value = 5;
                        else
                           value = ((dist > 0) ? 4 : 3);
                     }
                  }
               }
            break;  /* full-ring */
      }  /* switch */

   return(mkchnl(&(node[n].port[value]), ((value < portnum) ? 2 : 1), msg));
}     /* route */

/* NORMAL -- normal is taken from D. Knuth's The Art of Computer Programming,
Seminumerical Algorithms. */

      float normal()
      { float r, r2, x;
        int   i, val;
        r = 0.0;
        for (i=1; i<=12; i++) r =+ (rand() / 32767.0);
printf("normal:r:%f#",r);
        r = (r - 6) / 4.0;
        r2 = r * r;
        x = ((((r2 * 0.029899776 + 0.008355968) * r2 + 0.076542912)
            * r2 + 0.252408784) * r2 + 3.949846138) * r;
printf("normal:x:%f#",x);
        return(x);
      }  /* normal */


/* MINDIST -- mindist is used by the full-ring routing algorithm to find
the lesser horizontal distance between nodes. */

mindist(t, nd)
  int t, nd;
  { register int m, s, value;
    s = t - nd;
    m = ((s > 0) ? n : t);
    while (higher(--m, nd) == 0);
    m++;
    if (s > 0)
       value = ((s <= m / 2) ? s : (s - m));
    else
       value = ((s > (-m / 2)) ? s : (m + s));
    return(value);
  }   /* mindist */
```

```
#include "s4h"

/* PASS2 -- pass two is called by run for those ports having been marked
by pass one as sending.  For each node either or both of the input and
output ports (and CPU output port for port number "portnum") may be
sending.  First, the output port's sending flag is checked; if set,
and if the "sending channel number" bit is set, no byte is moved.
If the "sending channel number" bit is not set, a byte is moved
if there is room in the inbuf of that channel, otherwise,
the reject bit is set.  If the port is to the CPU and the byte was the
last in the message, an arrival trace message is output.  If the message
was also a teardown, the channel is removed.
        The input port is processed next, according to the channel flag.
If the channel is waiting for the controller, the byte is rejected.  If
it is sending to the controller, the byte is moved into the controller's
buffer; if it was the last of the address bytes, the message is routed.
Then, if a channel is available to be placed at the selected output port,
it is done, and the channel's flag is set to "sending to output port"; if
no channel is available, the flag is set to "waiting for channel".
        If the flag indicates "waiting for channel", the message is routed
again and another attempt made to place a channel.  If successful, the
flag is changed to "sending to output port".  If the flag indicates
"sending to output", a byte is moved if there is room at the "outbuf"
of the channel indicated by the sending channel's "destc".
        Finally, if the port is to the CPU, the CPU output is processed
similarly to a regular output port.
*/


pass2(port)
  register struct porttype *port;
  { char *flags, *flagp, *buffer;
    register struct chnltype *chnl;
    struct msgtype *msg;
    register struct nodetype *pnode;
    struct msdstype *mdesc;

    pnode = port->prnode;

    /* process output port */

    if (port->pflag & 0002)
    { chnl = port->outchnl;
      if (*(flagp = &(port->pflag)) & 0010)     /* sending channel address */
        (*flagp) =& 0367;
      else                      /* sending data */
      { if (p < portnum)        /* non-cpu port */
        { if (chnl->inbuf < inbuflim)   /* room to receive */
          { (chnl->inbuf)++;
            (chnl->outbuf)--;
            printmsg('s',1);
          }
          else                  /* no room -- reject */
          { (*flagp) =| 0004;
            printmsg('f',0);
          }
```

```
        }               /* non-cpu port */

        else    /* cpu port */
        { chnl->outbuf --;
          printmsg('s',1);
          if (++(chnl->inbuf) >= (msg = chnl->cmsg)->length) /* finished */
          { if (msg->kind)      /* end of teardown */
              rmchnl(port, 0);
            else                        /* end of regular message */
              chnl->inbuf = 0;
            printmsg('a',msg->source,msg->stime);
            msg->nextm = freemsg;       /* return message to free list */
            freemsg = msg;
          }     /* message finished */
        }               /* cpu port */
      }
    }   /* output port */

    /* process input port */

    if (port->pflag & 0020)
    { chnl = port->inchnl;
      switch(chnl->cflag)
      {
        case 0: /* waiting for controller -- reject */
                port->pflag =| 0040;
                printmsg('f',2);
                break;
        case 1: /* sending to controller */
                (chnl->inbuf)--;
                printmsg('s',3);
                if(++(pnode->conbuf) >= (msg = chnl->cmsg)->adleng)
                { chnl->inbuf =+ pnode->conbuf;
                  printmsg('g',0);
                  pnode->conbuf = 0;
                  if (chnl->destc = route(msg))
                    chnl->cflag = 2;
                  else  /* no chnl to send to -- wait */
                    chnl->cflag = 3;
                  pnode->nflag =& 0373;
                }
                break;
        case 2: /* sending to output port or CPU */
                if((chnl->destc) &&
                   (*(buffer = &(chnl->destc->outbuf)) < outbuflim))
                { (*buffer)++;
                  (chnl->inbuf)--;
                  printmsg('s',0);
                  if ((chnl->cmsg->kind) && (chnl->inbuf == 0) &&
                      (chnl->outbuf == 0))
                  { chnl->destc->cmsg = chnl->cmsg;
                    rmchnl(port, 1);
                  }
                }
                else
                { port->pflag =| 0040;  /* reject */
```

```
                    printmsg('f',1);
                  }
                  break;
          case 3: /* waiting for channel to be created at output */
                  if (chnl->destc = route(chnl->cmsg))
                    chnl->cflag = 2;      /* successful */
                  else    /* still no available channels -- reject */
                  { port->pflag =| 0040;
                    printmsg('f',1);
                  }
                  break;
        } /* switch */
      }   /* input port */

      /* output from cpu */

      if ((p == portnum) && (port->pflag & 0100))
      { if (pnode->nflag & 0020)  /* sending channel number */
          pnode->nflag =& 0357;
        else
        { if ((chnl = pnode->cpoutchnl)->inbuf < inbuflim)
          { (chnl->inbuf)++;
            (chnl->outbuf)--;
            printmsg('s',2);
            (pnode->cpbytes)++;
          }
          else
          { pnode->pflag =| 0010;
            printmsg('f',3);
            printmsg('b',pnode->cpbytes);
            pnode->cpbytes = 0;
          }
        }
      }   /* output from cpu */
    }     /* pass2 */


/* MKCHNL -- makechannel takes a pointer to a port, an option number, and a
pointer to a message and creates a channel at that port for the message.  If
the option "kind" is 0, the channel is added to those from the CPU, after that
indicated by "cpoutchnl".  If kind is 1, the channel is an output to the CPU.
If it is 2, the channel is an output to another node.  mkchnl increments the
appropriate channel count and prints a message with the number of active
channels. */

struct chnltype *mkchnl(port, kind, msg)
  register struct porttype *port;
  char kind;
  struct msgtype *msg;
  { register struct chnltype *nchnl, *ochnl;
    struct chnltype *head, *tail;
    register struct nodetype *pnode;

    if (nchnl = freechnl)       /* get next avail free channel */
    { nchnl->cmsg = msg;
      freechnl = freechnl->nextc;
```

```
nchnl->outbuf = nchnl->inbuf = nchnl->cflag = nchnl->destc = 0;
switch(kind)
{ case 0: /* output from CPU */
    if (ochnl = (pnode = port->prnode)->cpoutchnl)
    { /* already some active channels */
      (nchnl->nextc = ochnl->nextc)->lastc = nchnl;
      (nchnl->lastc = ochnl)->nextc = nchnl;
    }
    else
    { /* this will be only channel */
      pnode->cpoutchnl = port->inchnl = nchnl;
      nchnl->nextc = nchnl->lastc = nchnl;
      pnode->nflag =| 0020;
    }
    (pnode->cpchnls)++;
    nchnl->outbuf = msg->length;

    /* queue channel for controller */
    if (pnode->waithead)
    { /* already some queued for controller */
      pnode->waittail->destc = nchnl;
      pnode->waittail = nchnl;
    }
    else
      pnode->waithead = pnode->waittail = nchnl;
    break;
  case 1: /* output port to CPU */
    if (ochnl = port->outchnl)
    { (nchnl->nextc = ochnl->nextc)->lastc = nchnl;
      (nchnl->lastc = ochnl)->nextc = nchnl;
    }
    else
    { port->outchnl = nchnl->nextc = nchnl->lastc = nchnl;
      port->pflag =| 0010;
    }
    (port->chnls)++;
    break;
  case 2: /* output port to another node */
    if (ochnl = port->outchnl)
    { (nchnl->nextc = ochnl->nextc)->lastc = nchnl;
      (nchnl->lastc = ochnl)->nextc = nchnl;
    }
    else
    { port->outchnl = port->link->inchnl = nchnl;
      nchnl->nextc = nchnl->lastc = nchnl;
      port->pflag =| 0010;
    }
    (port->chnls)++;
    if ((pnode = port->link->prnode)->waithead)
    { /* already some queued */
      pnode->waittail->destc = nchnl;
      pnode->waittail = nchnl;
    }
    else
      pnode->waithead = pnode->waittail = nchnl;
    break;
```

```
      } /* switch */
      return(nchnl);
    }
    else        /* no channel free */
    { printmsg('n',1);
      return(0);
    }
  }     /* mkchnl */

/* RMCHNL -- remove channel takes a pointer to a port and an option number and
removes a channel from that port, and decrements the channel count.  If kind is
0, the channel was to the CPU.  If it is 1, the global port number 'p' is
checked to see if the channel was from anotner node or from the CPU. */

rmchnl(port, kind)
  register struct porttype *port;
  char kind;
  { register struct chnltype *chnl;
    register struct porttype *oport;
    struct nodetype *pnode;

    if (kind)
    { /* not to cpu */
      chnl = port->inchnl;
      if ( p < portnum)
      { /* from external input port */
        if (--((oport = port->link)->chnls) == 0)
          port->inchnl = port->link->outchnl = 0;
        else
        { chnl->nextc->lastc = chnl->lastc;
          chnl->lastc->nextc = chnl->nextc;
          if (oport->outchnl == chnl)
            oport->outchnl = chnl->nextc;
        }
      }
      else      /*from cpu */
      { if (--((pnode = port->prnode)->cpchnls) == 0)
          port->inchnl = port->prnode->cpoutchnl = 0;
        else
        { chnl->nextc->lastc = chnl->lastc;
          chnl->lastc->nextc = chnl->nextc;
          if (pnode->cpoutchnl == chnl)
            pnode->cpoutchnl = chnl->nextc;
        }
      }
    }
    else        /* to cpu */
    { chnl = port->outchnl;
      if (--(port->chnls) == 0)
        port->outchnl = 0;
      else
      { chnl->nextc->lastc = chnl->lastc;
        chnl->lastc->nextc = chnl->nextc;
      }
    }
```

```
    chnl->nextc = freechnl;
    freechnl = chnl;
}       /* rmchnl */
```

```
#include "s4h"

/* OTHERPORT -- otherport uses the global node and port (n & p) and returns
a pointer to the port attached to the specified port.  The first operation
performed is finding the number of the other node.  A pointer to the port
can then be found simply.  If there is no node attached or if p is not valid
for the structure, zero is returned. */

struct porttype *otherport()
  { register int on;

    if (p == portnum)   /* if CPU port, return 1 */
      return(1);
    if (on = othernode())
    { switch(shape)
      { case 1: /* simple binary */
                switch(p)
                { case 0: return(&(node[on].port[((n & 0001) ? 2 : 1)]));
                  case 1: case 2:
                          return(&(node[on].port[0]));
                  default: return(0);
                }
        case 2:         /* full-ring */
                switch(p)
                { case 0: return(&(node[on].port[((n & 0001) ? 2 : 1)]));
                  case 1: case 2:
                          return(&(node[on].port[0]));
                  case 3: return(&(node[on].port[4]));
                  case 4: return(&(node[on].port[3]));
                  default: return(0);
                }
        case 3: /* half-ring */
        case 4: /* user-defined 4th link */
                switch(p)
                { case 0: return(&(node[on].port[((n & 0001) ? 2 : 1)]));
                  case 1: case 2:
                          return(&(node[on].port[0]));
                  case 3: return(&(node[on].port[3]));
                  default: return(0);
                }
      } /* switch (shape) */
    }
    else
      return(0);
  }     /* otherport */


/* OTHERNODE -- othernode takes the global values n (node) and p (port) and
obtains the number of the node attached to the specified port.  If the number
is valid (at least 1 and at most "nodenum") it is returned; otherwise zero
is returned, indicating that the specified port is attached to a non-existent
node.  Most of the values are obtained by a simple computation. */

othernode()
  { register int val;
```

```
switch(shape)
{ case 1:   /* simple binary */
            switch(p)
            { case 0: val = n / 2;  break;
              case 1: val = n * 2;  break;
              case 2: val = (n * 2) + 1;  break;
              default: val = 0;  break;
            }
            break;
  case 2:   /* full-ring */
            switch(p)
            { case 0: val = n / 2;  break;
              case 1: val = n * 2;  break;
              case 2: val = (n * 2) + 1;  break;
              case 3: val = n - 1;
                      if (higher(val, n))
                      val   = (n * 2) - 1;
                      break;
              case 4: val = n + 1;
                      if (higher(n, val))
                        val   = (n + 1) / 2;
                      break;
              default: val = 0;  break;
            }
            break;
  case 3:   /* half-ring*/
            switch(p)
            { case 0: val = n / 2;  break;
              case 1: val = n * 2;  break;
              case 2: val = (n * 2) + 1;  break;
              case 3: shape = 2;
                      p = ((n & 0001) ? 4 : 3);
                      val = othernode();
                      p = 3;
                      shape = 3;
                      break;
              default: val = 0;  break;
            }
            break;
  case 4:   /* user-defined 4th link */
            switch(p)
            { case 0: val = n / 2;  break;
              case 1: val = n * 2;  break;
              case 2: val = (n * 2) + 1;  break;
              case 3: if (prompt)
                      printf("Node   %d is attached to node:",n);
                      val = getnum();
                      break;
              default: val = 0;  break;
            }
            break;
 }   /* switch (shape) */
 return(((val <= nodenum) && (val >= 1)) ? val : 0);
}    /* othernode */
```

```
#include "s4h"
/* PRINTMSG */

 printmsg(k, a, b)
 char k;
 register int a, b;

   { switch(output)
     { case 1:
               if (k == 't')
                 printf("time is %d#",tick);
               else
               { if (k == 'a')
                   printf("Message arrives at %d from %d since %d#",n,a,b);
                 else
                 { switch(k)
                   { case 'b': printf("Bytes sent "); break;
                     case 'e': printf("End of run "); break;
                     case 'f': printf("Buffer full, byte rejected ");
                               break;
                     case 'g': printf("Start new channel "); break;
                     case 'l': printf("Link contention "); break;
                     case 'n': printf("No more items "); break;
                     case 'q': printf("Channel created "); break;
                     case 'r': printf("Start of run "); break;
                     case 's': printf("One byte sent "); break;
                     case 'x': printf("End of session "); break;
                     default:  printf("%c ",k); break;
                   }
                   printf("at node %d, port %d: %d#",n,p,a);
                 }
               }
               break;
       case 2:
               if (k == 't')
                 printf("%c %d 0 0#",k,tick);
               else
               { if (k == 'a')
                   printf("%c %d %d %d#",k,n,a,b);
                 else
                   printf("%c %d %d %d#",k,n,p,a);
               }
               break;
     }   /* switch */
   }  /* printmsg */

/* GETNUM */

getnum()
  { register int m;
    m = 0;
    while (((c = getchar()) >= '0') && (c <= '9'))
      m = (m * 10) + (c - '0');
    return(m);
  }   /* getnum */
```

```
/* HIGHER -- higher takes two node numbers (a & b) and returns true if
a is on a higher level in the tree (closer to the root) than b). */

higher(a,b)
  register int a, b;
  { while ((a != 0) && (b != 0))
    { a =/ 2;
      b =/ 2;
    }
    return((a == 0) && (b != 0));
  }     /* higher */
```

```
#include "s4h"
dump()
{ int i,j;
  printf(" Dump state#");
  for (i=1;i<=nodenum;i++)
  { struct nodetype *pnode;
    struct porttype *pport;
    pnode = &(node[i]);
    printf("#Node %d nflag:%o conbuf:%d#",i,pnode->nflag,pnode->conbuf);
    printf(" cpchnls:%d waithead:%d waittail:%d#",
            pnode->cpchnls,pnode->waithead,pnode->waittail);
    if (pnode->msgdesc)
      printf("Message descriptor:%d#",pnode->msgdesc);
    if (pnode->cpoutchnl)
      printchnl(pnode->cpoutchnl);
    for (j=0;j<=portnum;j++)
    { pport = &(pnode->port[j]);
      printf("  Port %d pflag:%o#", j, pport->pflag);
      printf("  qhead:%d qtail:%d link:%d#",
              pport->qhead, pport->qtail, pport->link);
      if (pport->inchnl)
      { printf("Inchannel:");
        printchnl(pport->inchnl);
      }
      if (pport->outchnl)
      { printf("Outchannel:");
        printchnl(pport->outchnl);
      }
    }
  }
}

printchnl(chnl)
struct chnltype *chnl;
{ printf("Channel:%d has:#",chnl);
  printf("    nextc:%d lastc:%d destc:%d cmsg:%d#",chnl->nextc,chnl->lastc,
          chnl->destc, chnl->cmsg);
  printf("    outbuf:%d inbuf:%d cflag:%o#",chnl->outbuf,chnl->inbuf,
          chnl->cflag);
}
```

# APPENDIX C

## STATISTICS PROGRAM CODE

```
#define DISTMAX 15
#define NODEMAX 128
#define PORTMAX 6
#define CHNLMAX 10
#define LENGTHMAX 10

/* structure declaratins */

struct timetype { float timesum;
                  int msgnum;
                };

/* variable declarations */

struct timetype time[DISTMAX];
int     bytes[LENGTHMAX],
        chnls[NODEMAX][PORTMAX],
        active[CHNLMAX],
        tick, num, overflow,
        a, b, c,
        nodenum, portnum, shape, num;
char    kind;

/* MAIN -- each time this routine gets an 'r' trace message, it will start
a new run.  It first gets the parameters output by 'initialize' in the
simulator and then gets the trace output by 'run'. */

main()
{
  while ((kind = gettrace()) == 'r')
  { getparams();
    run();
  }
}       /* main */


/* GETPARAMS -- this routine dovetails with the parameters echoed by 'init'.
It is not fault-tolerant -- if 'init' detects bad inputs and prints out
error messages, 'getparams' will not be able to handle them */

getparams()
{

  /* get output format and discard -- assumed to be abbreviated */

  printf("%d#",getnum());

  /* get debugging level and discard */

  printf("%d#",getnum());

  /* get input and output buffer sizes and discard */

  printf("%d#",getnum());
  printf("%d#",getnum());
```

```
  /* get input channel selection option and discard */

  printf("%d#", getnum());

  /* get number of nodes */

  nodenum = getnum();
  printf("%d#",nodenum);

  /* get structure type */

  shape = getnum();
  printf("%d#",shape);

  switch(shape)
  { case 1: portnum = 3; break;
    case 2: portnum = 5; break;
    case 3: case 4: portnum = 4; break;
  }

  /* ignore characters indicating nodes and links killed */

  while (putchar(getchar()) != '#')
    while (putchar(getchar()) != '#') ;

  /* get routing algorithm */

  printf("%d#",getnum());

  /* ignore message descriptor definition */

  while (putchar(getchar()) != '#')
    while (putchar(getchar()) != '#') ;

  /* ignore length of run */

  printf("%d#",getnum());
}       /* initialize */

/* RUN -- this initializes the data structures and then reads in the
trace and updates the structures.  Upon encountering an 'e' message,
it processes the data and outputs the results. */

run()
{ register int i,j;

  for (i=0;i<DISTMAX;i++)
  { time[i].timesum = 0;
    time[i].msgnum  = 0;
  }
  for (i=0;i<LENGTHMAX;i++)
    bytes[i] = 0;
  for (i=1;i<=nodenum;i++)
    for (j=0;j<=portnum;j++)
      chnls[i][j] = 0;
```

```
overflow = 0;

/* get trace messages up to and including 'e' -- end of run */

while (gettrace() != 'e')
{ switch(kind)
  { /* depending upon kind of message, update structures appropriately */

    case 'a': /* arrival of message at destination.  'a' has the
                 destination node number (ignored), 'b' has the path length
                 in switches, and 'c' has the start time. */
              time[b].timesum =+ (tick - c);
              (time[b].msgnum)++;
              break;
    case 'b': /* reject terminates transmission of string of bytes. 'a'
                 has sending node (ignored), 'b' has port (ignored), and
                 'c' has the number of bytes, without channel numbers. */
              bytes[c]++;
              break;
    case 'g': /* message routed and channel created. 'a' has node, 'b' has
                 port, and 'c' has total channels there. */
              chnls[a][b]++;
              break;
    case 'n': /* exhaustion of some free list. 'a' and 'b' are node and port
                 and 'c' is kind of list. Print warning message. */
              printf("*** List of type %d exhausted#",c);
              printf("  at node %d port %d time %d#",a,b,tick);
              break;
    case 't': /* start of new tick. 'a' has tick number; ignore others. */
              tick = a;
              break;
    default:  /* all other messages ignored */
              break;
  }   /* switch(kind) */
}     /* while not end of run */

/* summarize statistics and print results */

/* for each tranmission distance, print average time and number of messages */

printf("Distance      average time      number of messages#");
for (i=0;i<DISTMAX;i++)
  printf("  %d              %f                    %d#",
         i, (time[i].timesum / time[i].msgnum), time[i].msgnum);
printf("##");

/* for each block length, print number of times that many bytes were sent */

printf("Number of bytes sent      Number of occurrences#");
for(i=0;i<LENGTHMAX;i++)
  printf("    %d                      %d#",i,bytes[i]);
printf("##");

/* for each number of active channels, from zero up, print the number of
   ports having that number */
```

```
  for (i=1;i<=nodenum;i++)
    for (j=0;j<=portnum;j++)
    { if ((num = chnls[i][j]) < CHNLMAX)
        active[num]++;
      else
        overflow++;
    }
  printf("Number of active channels      Number of occurrences#");
  for (i=0;i< CHNLMAX;i++)
    printf("       %d                              %d#", i, active[i]);

  printf("More than %d channels:  %d occurrences#", CHNLMAX - 1, overflow);

  printf("End of summary#");
}       /* run */


/* GETTRACE -- reads in one standard trace.  Puts first (char) field
in 'kind' and next three (int) fields in 'a, 'b', and 'c'.  Returns 'kind'*/

gettrace()
{ kind = getchar();
  getchar();
  a = getnum();
  b = getnum();
  c = getnum();
  return(kind);
}       /* gettrace */

/* GETNUM -- this reads in an integer and reads the following non-digit */

getnum()
{ register int m;
  register char c;

  m = 0;
  while(((c = getchar()) >= '0') && (c <= '9'))
    m = (m * 10) + (c - '0');
  return(m);
}       /* getnum */
```